```python
def execute_example(fn, args=[]):
    if __name__ == "__main__" and RUN_EXAMPLES:
        fn(*args)


class DummyOptimizer(torch.optim.Optimizer):
    def __init__(self):
        self.param_groups = [{"lr": 0}]
        None

    def step(self):
        None

    def zero_grad(self, set_to_none=False):
        None


class DummyScheduler:
    def step(self):
        None
```

> My comments are blockquoted. The main text is all from the paper its

Background

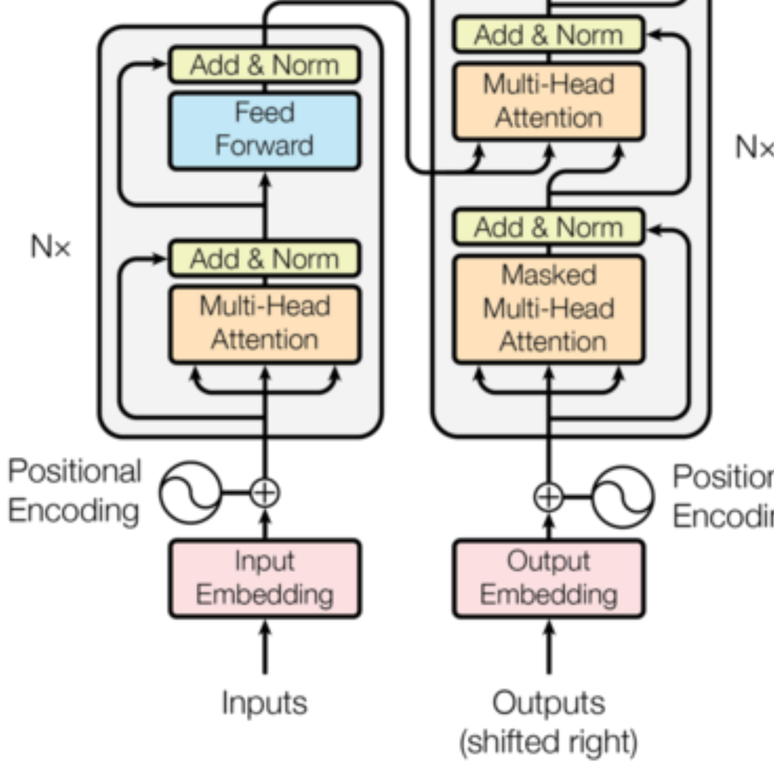
The goal of reducing sequential computation also forms the foundation of t ByteNet and ConvS2S, all of which use convolutional neural networks as b computing hidden representations in parallel for all input and output positi number of operations required to relate signals from two arbitrary input or distance between positions, linearly for ConvS2S and logarithmically for B difficult to learn dependencies between distant positions. In the Transforme number of operations, albeit at the cost of reduced effective resolution due weighted positions, an effect we counteract with Multi-Head Attention.

Model Architecture

Most competitive neural sequence transduction models have an encoder-de
the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n$
continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Given $\mathbf{z}$, the decoder then g
$(y_1, \ldots, y_m)$ of symbols one element at a time. At each step the model is a
consuming the previously generated symbols as additional input when gene

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this an
    other models.
    """

    def __init__(self, encoder, decoder, src_embed, tgt_embed
        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_ma

    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_
```

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."

    def __init__(self, d_model, vocab):
```



Encoder and Decoder Stacks

Encoder

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

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

```
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
```

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

That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x$ the function implemented by the sub-layer itself. We apply dropout (cite) 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 produce outputs of dimension $d_{\text{model}} = 512$.

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to
    """

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

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

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (define

    def __init__(self, size, self_attn, feed_forward, dropout
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
```

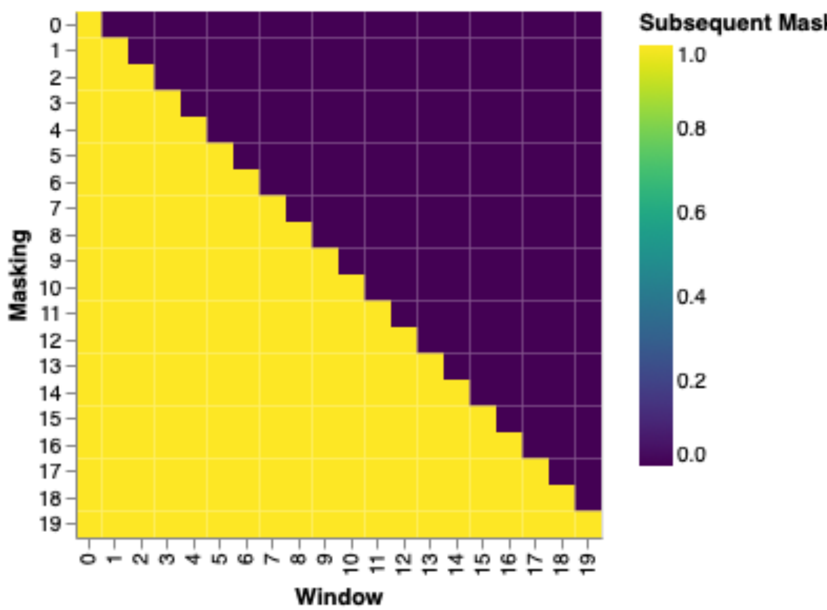
```
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    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)
```

In addition to the two sub-layers in each encoder layer, the decoder inserts
performs multi-head attention over the output of the encoder stack. Simila
residual connections around each of the sub-layers, followed by layer norm

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward

    def __init__(self, size, self_attn, src_attn, feed_forwar
        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, dropo

    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 = self.sublayer[1](x, lambda x: self.src_attn(x, m,
        return self.sublayer[2](x, self.feed_forward)
```

We also modify the self-attention sub-layer in the decoder stack to prevent
subsequent positions. This masking, combined with fact that the output em
position, ensures that the predictions for position i can depend only on the
less than i.

```
                {
                    "Subsequent Mask": subsequent_mask(20)[0]
                    "Window": y,
                    "Masking": x,
                }
            )
            for y in range(20)
            for x in range(20)
        ]
    )

    return (
        alt.Chart(LS_data)
        .mark_rect()
        .properties(height=250, width=250)
        .encode(
            alt.X("Window:O"),
            alt.Y("Masking:O"),
            alt.Color("Subsequent Mask:Q", scale=alt.Scale(sc
        )
        .interactive()
    )


show_example(example_mask)
```



Attention

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

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

```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)) / mat
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(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 (c
(multiplicative) attention. Dot-product attention is identical to our algorith
factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function usin
with a single hidden layer. While the two are similar in theoretical comple
much faster and more space-efficient in practice, since it can be implement
matrix multiplication code.

While for small values of d_k the two mechanisms perform similarly, additi
product attention without scaling for larger values of d_k (cite). We suspect
the dot products grow large in magnitude, pushing the softmax function in
extremely small gradients (To illustrate why the dot products get large, assu
q and k are independent random variables with mean 0 and variance 1. Th
$k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance d_k.). To counteract this effect,
$\frac{1}{\sqrt{d_k}}$.

Multi-head attention allows the model to jointly attend to information fro
subspaces at different positions. With a single attention head, averaging inh

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{he}$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW^Q_i, KW^K_i, \mathrm{V}$$

Where the projections are parameter matrices $W^Q_i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W^K_i \in$
$\mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each
$d_v = d_{\text{model}}/h = 64$. Due to the reduced dimension of each head, the t
similar to that of single-head attention with full dimensionality.

```
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)
```

```python
        del key
        del value
        return self.linears[-1](x)
```

Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways: 1) In "
layers, the queries come from the previous decoder layer, and the memory
the output of the encoder. This allows every position in the decoder to atte
input sequence. This mimics the typical encoder-decoder attention mechan
sequence models such as (cite).

2. The encoder contains self-attention layers. In a self-attention layer all queries come from the same place, in this case, the output of the previ encoder. Each position in the encoder can attend to all positions in the encoder.

3. Similarly, self-attention layers in the decoder allow each position in th all positions in the decoder up to and including that position. We need information flow in the decoder to preserve the auto-regressive prope inside of scaled dot-product attention by masking out (setting to $-\infty$ of the softmax which correspond to illegal connections.

Position-wise Feed-Forward Networks

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

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

While the linear transformations are the same across different positions, the
from layer to layer. Another way of describing this is as two convolutions w
dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer

and output tokens to vectors of dimension d_{model}. We also use the usual le
and softmax function to convert the decoder output to predicted next-toke
we share the same weight matrix between the two embedding layers and th
transformation, similar to (cite). In the embedding layers, we multiply thos

```
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)
```

Positional Encoding

Since our model contains no recurrence and no convolution, in order for th
order of the sequence, we must inject some information about the relative
tokens in the sequence. To this end, we add "positional encodings" to the i
bottoms of the encoder and decoder stacks. The positional encodings have t
the embeddings, so that the two can be summed. There are many choices o
learned and fixed (cite).

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

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

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

where pos is the position and i is the dimension. That is, each dimension c
corresponds to a sinusoid. The wavelengths form a geometric progression fr
chose this function because we hypothesized it would allow the model to e
relative positions, since for any fixed offset k, PE_{pos+k} can be represente

```
        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 + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)
```

> Below the positional encoding will add in a sine wave based on positio
> offset of the wave is different for each dimension.

```
def example_positional():
    pe = PositionalEncoding(20, 0)
    y = pe.forward(torch.zeros(1, 100, 20))

    data = pd.concat(
        [
            pd.DataFrame(
                {
                    "embedding": y[0, :, dim],
                    "dimension": dim,
                    "position": list(range(100)),
                }
            )
            for dim in [4, 5, 6, 7]
        ]
    )

    return (
        alt.Chart(data)
        .mark_line()
        .properties(width=800)
        .encode(x="position", y="embedding", color="dimension
        .interactive()
    )
```

We also experimented with using learned positional embeddings (cite) inst
versions produced nearly identical results. We chose the sinusoidal version
model to extrapolate to sequence lengths longer than the ones encountered

Full Model

> Here we define a function from hyperparameters to a full model.

```
def make_model(
    src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, d
):
    "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
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff)
        nn.Sequential(Embeddings(d_model, src_vocab), c(posit
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(posit
        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
```

Inference:

> Here we make a forward step to generate a prediction of the model. W
> transformer to memorize the input. As you will see the output is rando

```
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.data[0]
        ys = torch.cat(
            [ys, torch.empty(1, 1).type_as(src.data).fill_(ne
        )

    print("Example Untrained Model Prediction:", ys)


def run_tests():
    for _ in range(10):
        inference_test()


show_example(run_tests)
```

```
Example Untrained Model Prediction: tensor([[0, 0, 0, 0, 0, 0, 0
Example Untrained Model Prediction: tensor([[0, 3, 4, 4, 4, 4, 4
Example Untrained Model Prediction: tensor([[ 0, 10, 10, 10,  3,
Example Untrained Model Prediction: tensor([[ 0,  4,  3,  6, 10,
Example Untrained Model Prediction: tensor([[ 0,  9,  0,  1,  5,
Example Untrained Model Prediction: tensor([[ 0,  1,  5,  1, 10,
Example Untrained Model Prediction: tensor([[ 0,  1, 10,  9,  9,
Example Untrained Model Prediction: tensor([[ 0,  3,  1,  5, 10,
Example Untrained Model Prediction: tensor([[ 0,  3,  5, 10,  5,
Example Untrained Model Prediction: tensor([[0, 5, 6, 2, 5, 6, 2
```

Part 2: Model Training

Training

This section describes the training regime for our models.

> We stop for a quick interlude to introduce some of the tools needed to
> decoder model. First we define a batch object that holds the src and tar

```
        "Create a mask to hide padding and future words."
        tgt_mask = (tgt != pad).unsqueeze(-2)
        tgt_mask = tgt_mask & subsequent_mask(tgt.size(-1)).t
            tgt_mask.data
        )
        return tgt_mask
```

> Next we create a generic training and scoring function to keep track of
> generic loss compute function that also handles parameter updates.

Training Loop

```
class TrainState:
    """Track number of steps, examples, and tokens processed"

    step: int = 0  # Steps in the current epoch
    accum_step: int = 0  # Number of gradient accumulation st
    samples: int = 0  # total # of examples used
    tokens: int = 0  # total # of tokens processed
```

```
def run_epoch(
    data_iter,
    model,
    loss_compute,
    optimizer,
    scheduler,
    mode="train",
    accum_iter=1,
    train_state=TrainState(),
):
    """Train a single epoch"""
    start = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
```

```
                total_tokens += batch.ntokens
                tokens += batch.ntokens
                if i % 40 == 1 and (mode == "train" or mode == "train
                    lr = optimizer.param_groups[0]["lr"]
                    elapsed = time.time() - start
                    print(
                        (
                            "Epoch Step: %6d | Accumulation Step: %3d
                            + "| Tokens / Sec: %7.1f | Learning Rate:
                        )
                        % (i, n_accum, loss / batch.ntokens, tokens /
                    )
                    start = time.time()
                    tokens = 0
                del loss
                del loss_node
            return total_loss / total_tokens, train_state
```

Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consistin
sentence pairs. Sentences were encoded using byte-pair encoding, which ha
vocabulary of about 37000 tokens. For English-French, we used the signifi
English-French dataset consisting of 36M sentences and split tokens into a
vocabulary.

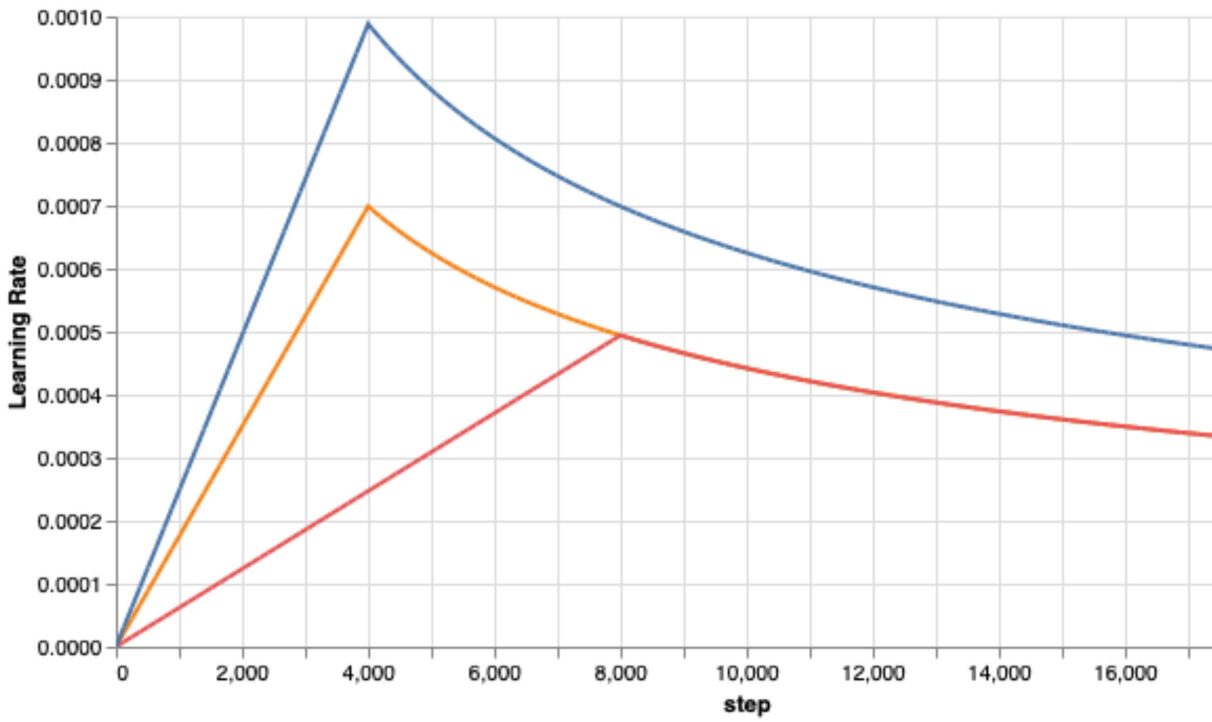
Sentence pairs were batched together by approximate sequence length. Eac
set of sentence pairs containing approximately 25000 source tokens and 250

Hardware and Schedule

We trained our models on one machine with 8 NVIDIA P100 GPUs. For
hyperparameters described throughout the paper, each training step took al
the base models for a total of 100,000 steps or 12 hours. For our big models
The big models were trained for 300,000 steps (3.5 days).

```python
def rate(step, model_size, factor, warmup):
    """
    we have to default the step to 1 for LambdaLR function
    to avoid zero raising to negative power.
    """
    if step == 0:
        step = 1
    return factor * (
        model_size ** (-0.5) * min(step ** (-0.5), step * war
    )



def example_learning_schedule():
    opts = [
        [512, 1, 4000],  # example 1
        [512, 1, 8000],  # example 2
        [256, 1, 4000],  # example 3
    ]

    dummy_model = torch.nn.Linear(1, 1)
    learning_rates = []

    # we have 3 examples in opts list.
    for idx, example in enumerate(opts):
        # run 20000 epoch for each example
        optimizer = torch.optim.Adam(
            dummy_model.parameters(), lr=1, betas=(0.9, 0.98)
        )
        lr_scheduler = LambdaLR(
            optimizer=optimizer, lr_lambda=lambda step: rate(
        )
        tmp = []
        # take 20K dummy training steps, save the learning ra
        for step in range(20000):
            tmp.append(optimizer.param_groups[0]["lr"])
            optimizer.step()
            lr_scheduler.step()
        learning_rates.append(tmp)
```

```
        return (
            alt.Chart(opts_data)
            .mark_line()
            .properties(width=600)
            .encode(x="step", y="Learning Rate", color="model_siz
            .interactive()
        )


    example_learning_schedule()
```



Regularization

Label Smoothing

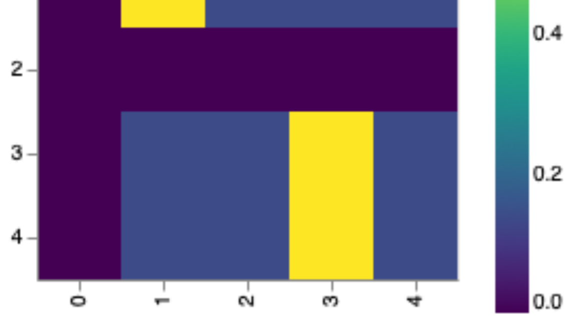
During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ (cite). T
model learns to be more unsure, but improves accuracy and BLEU score.

> We implement label smoothing using the KL div loss. Instead of using
> distribution, we create a distribution that has `confidence` of the correc
> `smoothing` mass distributed throughout the vocabulary.

```
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        return self.criterion(x, true_dist.clone().detach())
```

> Here we can see an example of how the mass is distributed to the word

```
# Example of label smoothing.


def example_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],
            [0, 0.2, 0.7, 0.1, 0],
            [0, 0.2, 0.7, 0.1, 0],
        ]
    )
    crit(x=predict.log(), target=torch.LongTensor([2, 1, 0, 3
    LS_data = pd.concat(
        [
            pd.DataFrame(
                {
                    "target distribution": crit.true_dist[x,
                    "columns": y,
                    "rows": x,
                }
            )
            for y in range(5)
            for x in range(5)
        ]
    )

    return (
```



> Label smoothing actually starts to penalize the model if it gets very con
> choice.

```
def loss(x, crit):
    d = x + 3 * 1
    predict = torch.FloatTensor([[0, x / d, 1 / d, 1 / d, 1 /
    return crit(predict.log(), torch.LongTensor([1])).data


def penalization_visualization():
    crit = LabelSmoothing(5, 0, 0.1)
    loss_data = pd.DataFrame(
        {
            "Loss": [loss(x, crit) for x in range(1, 100)],
            "Steps": list(range(99)),
        }
    ).astype("float")

    return (
        alt.Chart(loss_data)
        .mark_line()
        .properties(width=350)
        .encode(
            x="Steps",
            y="Loss",
        )
        .interactive()
    )
```

A First Example

> We can begin by trying out a simple copy-task. Given a random set of
> small vocabulary, the goal is to generate back those same symbols.

Synthetic Data

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

Loss Computation

```
class SimpleLossCompute:
    "A simple loss compute and train function."

    def __init__(self, generator, criterion):
        self.generator = generator
        self.criterion = criterion

    def __call__(self, x, y, norm):
        x = self.generator(x)
        sloss = (
            self.criterion(
                x.contiguous().view(-1, x.size(-1)), y.contig
            )
            / norm
        )
        return sloss.data * norm, sloss
```

```
            [ys, torch.zeros(1, 1).type_as(src.data).fill_(ne
        )
    return ys



# Train the simple copy task.


def example_simple_model():
    V = 11
    criterion = LabelSmoothing(size=V, padding_idx=0, smoothi
    model = make_model(V, V, N=2)

    optimizer = torch.optim.Adam(
        model.parameters(), lr=0.5, betas=(0.9, 0.98), eps=1e
    )
    lr_scheduler = LambdaLR(
        optimizer=optimizer,
        lr_lambda=lambda step: rate(
            step, model_size=model.src_embed[0].d_model, fact
        ),
    )

    batch_size = 80
    for epoch in range(20):
        model.train()
        run_epoch(
            data_gen(V, batch_size, 20),
            model,
            SimpleLossCompute(model.generator, criterion),
            optimizer,
            lr_scheduler,
            mode="train",
        )
        model.eval()
        run_epoch(
            data_gen(V, batch_size, 5),
            model,
            SimpleLossCompute(model.generator, criterion),
            DummyOptimizer(),
```

This task is much smaller than the WMT task considered in the paper, whole system. We also show how to use multi-gpu processing to make

Data Loading

> We will load the dataset using torchtext and spacy for tokenization.

```
# Load spacy tokenizer models, download them if they haven't
# downloaded already


def load_tokenizers():

    try:
        spacy_de = spacy.load("de_core_news_sm")
    except IOError:
        os.system("python -m spacy download de_core_news_sm")
        spacy_de = spacy.load("de_core_news_sm")

    try:
        spacy_en = spacy.load("en_core_web_sm")
    except IOError:
        os.system("python -m spacy download en_core_web_sm")
        spacy_en = spacy.load("en_core_web_sm")

    return spacy_de, spacy_en




def tokenize(text, tokenizer):
    return [tok.text for tok in tokenizer.tokenizer(text)]


def yield_tokens(data_iter, tokenizer, index):
    for from_to_tuple in data_iter:
        yield tokenizer(from_to_tuple[index])
```

```
    train, val, test = datasets.Multi30k(language_pair=("de",
    vocab_tgt = build_vocab_from_iterator(
        yield_tokens(train + val + test, tokenize_en, index=1
        min_freq=2,
        specials=["<s>", "</s>", "<blank>", "<unk>"],
    )

    vocab_src.set_default_index(vocab_src["<unk>"])
    vocab_tgt.set_default_index(vocab_tgt["<unk>"])

    return vocab_src, vocab_tgt


def load_vocab(spacy_de, spacy_en):
    if not exists("vocab.pt"):
        vocab_src, vocab_tgt = build_vocabulary(spacy_de, spa
        torch.save((vocab_src, vocab_tgt), "vocab.pt")
    else:
        vocab_src, vocab_tgt = torch.load("vocab.pt")
    print("Finished.\nVocabulary sizes:")
    print(len(vocab_src))
    print(len(vocab_tgt))
    return vocab_src, vocab_tgt


if is_interactive_notebook():
    # global variables used later in the script
    spacy_de, spacy_en = show_example(load_tokenizers)
    vocab_src, vocab_tgt = show_example(load_vocab, args=[spa
```

```
Finished.
Vocabulary sizes:
59981
36745
```

> Batching matters a ton for speed. We want to have very evenly divided
> minimal padding. To do this we have to hack a bit around the default t
> code patches their default batching to make sure we search over enoug
> batches.

```
        bs_id,
        torch.tensor(
            src_vocab(src_pipeline(_src)),
            dtype=torch.int64,
            device=device,
        ),
        eos_id,
    ],
    0,
)
processed_tgt = torch.cat(
    [
        bs_id,
        torch.tensor(
            tgt_vocab(tgt_pipeline(_tgt)),
            dtype=torch.int64,
            device=device,
        ),
        eos_id,
    ],
    0,
)
src_list.append(
    # warning - overwrites values for negative values
    pad(
        processed_src,
        (
            0,
            max_padding - len(processed_src),
        ),
        value=pad_id,
    )
)
tgt_list.append(
    pad(
        processed_tgt,
        (0, max_padding - len(processed_tgt)),
        value=pad_id,
    )
)
```

```
def tokenize_en(text):
    return tokenize(text, spacy_en)

def collate_fn(batch):
    return collate_batch(
        batch,
        tokenize_de,
        tokenize_en,
        vocab_src,
        vocab_tgt,
        device,
        max_padding=max_padding,
        pad_id=vocab_src.get_stoi()["<blank>"],
    )

train_iter, valid_iter, test_iter = datasets.Multi30k(
    language_pair=("de", "en")
)

train_iter_map = to_map_style_dataset(
    train_iter
)  # DistributedSampler needs a dataset len()
train_sampler = (
    DistributedSampler(train_iter_map) if is_distributed
)
valid_iter_map = to_map_style_dataset(valid_iter)
valid_sampler = (
    DistributedSampler(valid_iter_map) if is_distributed
)

train_dataloader = DataLoader(
    train_iter_map,
    batch_size=batch_size,
    shuffle=(train_sampler is None),
    sampler=train_sampler,
    collate_fn=collate_fn,
)
valid_dataloader = DataLoader(
    valid_iter_map,
    batch_size=batch_size,
    shuffle=(valid_sampler is None),
```

```
    print(f"Train worker process using GPU: {gpu} for trainin
    torch.cuda.set_device(gpu)

    pad_idx = vocab_tgt["<blank>"]
    d_model = 512
    model = make_model(len(vocab_src), len(vocab_tgt), N=6)
    model.cuda(gpu)
    module = model
    is_main_process = True
    if is_distributed:
        dist.init_process_group(
            "nccl", init_method="env://", rank=gpu, world_siz
        )
        model = DDP(model, device_ids=[gpu])
        module = model.module
        is_main_process = gpu == 0

    criterion = LabelSmoothing(
        size=len(vocab_tgt), padding_idx=pad_idx, smoothing=0
    )
    criterion.cuda(gpu)

    train_dataloader, valid_dataloader = create_dataloaders(
        gpu,
        vocab_src,
        vocab_tgt,
        spacy_de,
        spacy_en,
        batch_size=config["batch_size"] // ngpus_per_node,
        max_padding=config["max_padding"],
        is_distributed=is_distributed,
    )

    optimizer = torch.optim.Adam(
        model.parameters(), lr=config["base_lr"], betas=(0.9,
    )
    lr_scheduler = LambdaLR(
        optimizer=optimizer,
        lr_lambda=lambda step: rate(
            step, d_model, factor=1, warmup=config["warmup"]
        ),
```

```
            GPUtil.showUtilization()
            if is_main_process:
                file_path = "%s%.2d.pt" % (config["file_prefix"],
                torch.save(module.state_dict(), file_path)
            torch.cuda.empty_cache()

            print(f"[GPU{gpu}] Epoch {epoch} Validation ====", fl
            model.eval()
            sloss = run_epoch(
                (Batch(b[0], b[1], pad_idx) for b in valid_datalo
                model,
                SimpleLossCompute(module.generator, criterion),
                DummyOptimizer(),
                DummyScheduler(),
                mode="eval",
            )
            print(sloss)
            torch.cuda.empty_cache()

        if is_main_process:
            file_path = "%sfinal.pt" % config["file_prefix"]
            torch.save(module.state_dict(), file_path)



    def train_distributed_model(vocab_src, vocab_tgt, spacy_de, s
        from the_annotated_transformer import train_worker

        ngpus = torch.cuda.device_count()
        os.environ["MASTER_ADDR"] = "localhost"
        os.environ["MASTER_PORT"] = "12356"
        print(f"Number of GPUs detected: {ngpus}")
        print("Spawning training processes ...")
        mp.spawn(
            train_worker,
            nprocs=ngpus,
            args=(ngpus, vocab_src, vocab_tgt, spacy_de, spacy_en
        )
```

```
        "file_prefix": "multi30k_model_",
    }
    model_path = "multi30k_model_final.pt"
    if not exists(model_path):
        train_model(vocab_src, vocab_tgt, spacy_de, spacy_en,

    model = make_model(len(vocab_src), len(vocab_tgt), N=6)
    model.load_state_dict(torch.load("multi30k_model_final.pt
    return model


if is_interactive_notebook():
    model = load_trained_model()
```

> Once trained we can decode the model to produce a set of translations. the first sentence in the validation set. This dataset is pretty small so the search are reasonably accurate.

Additional Components: BPE, S Averaging

> So this mostly covers the transformer model itself. There are four aspec explicitly. We also have all these additional features implemented in O

> 1. BPE/ Word-piece: We can use a library to first preprocess the da units. See Rico Sennrich's subword-nmt implementation. These n the training data to look like this:

▁Die ▁Protokoll datei ▁kann ▁ heimlich ▁per ▁E - Mail ▁oder ▁F ▁bestimmte n ▁Empfänger ▁gesendet ▁werden .

> 2. Shared Embeddings: When using BPE with shared vocabulary we weight vectors between the source / target / generator. See the (c

```
ps[0].copy_(torch.sum(*ps[1:]) / len(ps[1:]))
```

Results

On the WMT 2014 English-to-German translation task, the big transform in Table 2) outperforms the best previously reported models (including ens BLEU, establishing a new state-of-the-art BLEU score of 28.4. The config in the bottom line of Table 3. Training took 3.5 days on 8 P100 GPUs. Eve previously published models and ensembles, at a fraction of the training cos models.

On the WMT 2014 English-to-French translation task, our big model ach outperforming all of the previously published single models, at less than 1/ previous state-of-the-art model. The Transformer (big) model trained for I dropout rate Pdrop = 0.1, instead of 0.3.

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-a English-to-German and English-to-French newstest2014 tests at a fraction of the tr

Model	BLEU		Training Cost	
	EN-DE	EN-FR	EN-DE	
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		1
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	1
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	1
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	1
Deep-Att + PosUnk Ensemble [39]		40.4		8
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	1
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	1
Transformer (base model)	27.3	38.1		**3.3 · 1**
Transformer (big)	**28.4**	**41.8**		2.3 · 1

> With the addtional extensions in the last section, the OpenNMT-py r EN-DE WMT. Here I have loaded in those parameters to our reimple

```
# Load data and model for output checks
```

```
        ]
        tgt_tokens = [
            vocab_tgt.get_itos()[x] for x in rb.tgt[0] if x !
        ]

        print(
            "Source Text (Input)        : "
            + " ".join(src_tokens).replace("\n", "")
        )
        print(
            "Target Text (Ground Truth) : "
            + " ".join(tgt_tokens).replace("\n", "")
        )
        model_out = greedy_decode(model, rb.src, rb.src_mask,
        model_txt = (
            " ".join(
                [vocab_tgt.get_itos()[x] for x in model_out i
            ).split(eos_string, 1)[0]
            + eos_string
        )
        print("Model Output               : " + model_txt.rep
        results[idx] = (rb, src_tokens, tgt_tokens, model_out
    return results


def run_model_example(n_examples=5):
    global vocab_src, vocab_tgt, spacy_de, spacy_en

    print("Preparing Data ...")
    _, valid_dataloader = create_dataloaders(
        torch.device("cpu"),
        vocab_src,
        vocab_tgt,
        spacy_de,
        spacy_en,
        batch_size=1,
        is_distributed=False,
    )

    print("Loading Trained Model ...")
```

what is happening at each layer of the attention

```
def mtx2df(m, max_row, max_col, row_tokens, col_tokens):
    "convert a dense matrix to a data frame with row and colu
    return pd.DataFrame(
        [
            (
                r,
                c,
                float(m[r, c]),
                "%.3d %s"
                % (r, row_tokens[r] if len(row_tokens) > r el
                "%.3d %s"
                % (c, col_tokens[c] if len(col_tokens) > c el
            )
            for r in range(m.shape[0])
            for c in range(m.shape[1])
            if r < max_row and c < max_col
        ],
        # if float(m[r,c]) != 0 and r < max_row and c < max_c
        columns=["row", "column", "value", "row_token", "col_
    )


def attn_map(attn, layer, head, row_tokens, col_tokens, max_d
    df = mtx2df(
        attn[0, head].data,
        max_dim,
        max_dim,
        row_tokens,
        col_tokens,
    )
    return (
        alt.Chart(data=df)
        .mark_rect()
        .encode(
            x=alt.X("col_token", axis=alt.Axis(title="")),
            y=alt.Y("row_token", axis=alt.Axis(title="")),
            color="value",
```

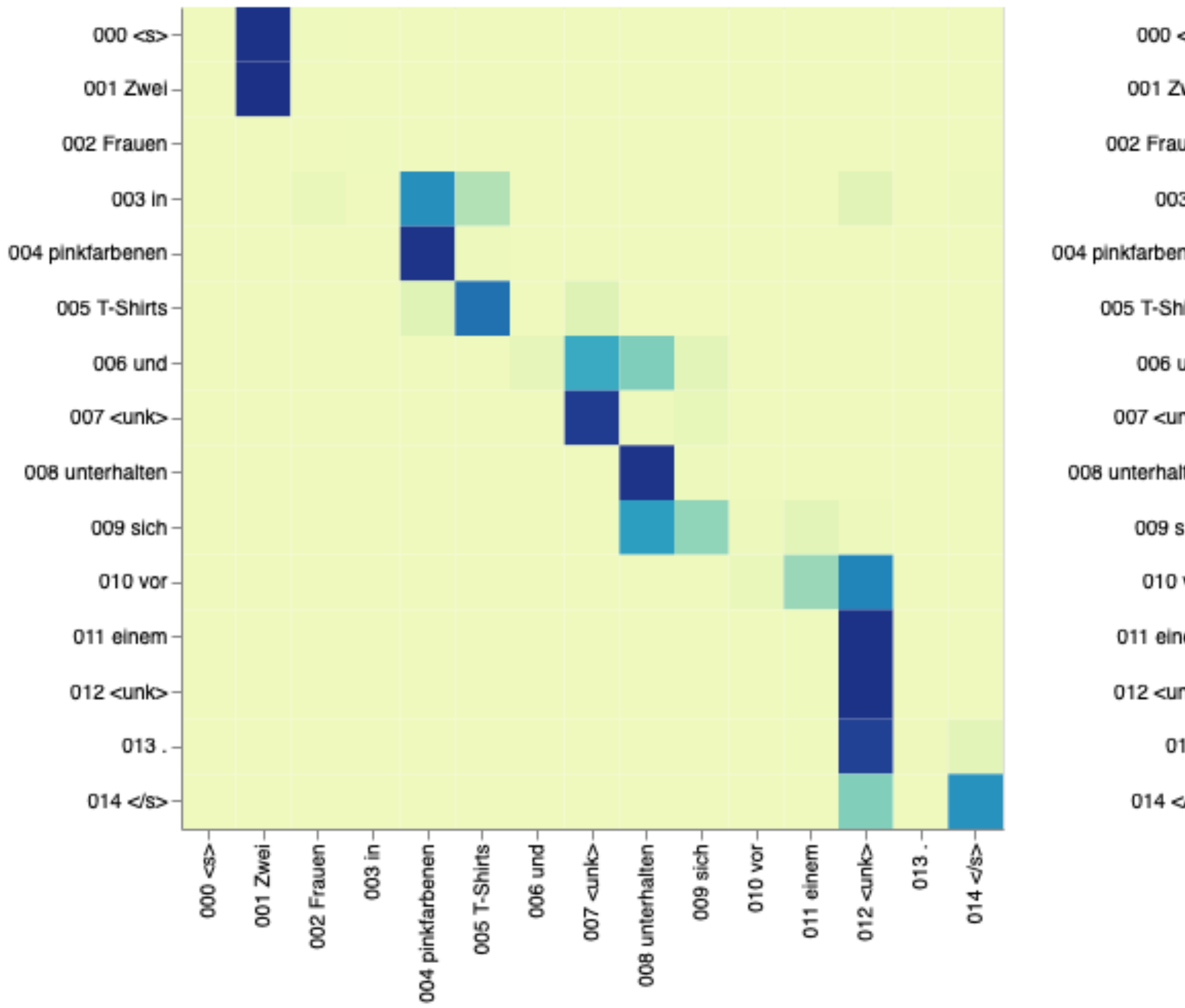
```python
def visualize_layer(model, layer, getter_fn, ntokens, row_tok
    # ntokens = last_example[0].ntokens
    attn = getter_fn(model, layer)
    n_heads = attn.shape[1]
    charts = [
        attn_map(
            attn,
            0,
            h,
            row_tokens=row_tokens,
            col_tokens=col_tokens,
            max_dim=ntokens,
        )
        for h in range(n_heads)
    ]
    assert n_heads == 8
    return alt.vconcat(
        charts[0]
        # | charts[1]
        | charts[2]
        # | charts[3]
        | charts[4]
        # | charts[5]
        | charts[6]
        # | charts[7]
        # layer + 1 due to 0-indexing
    ).properties(title="Layer %d" % (layer + 1))
```

Encoder Self Attention

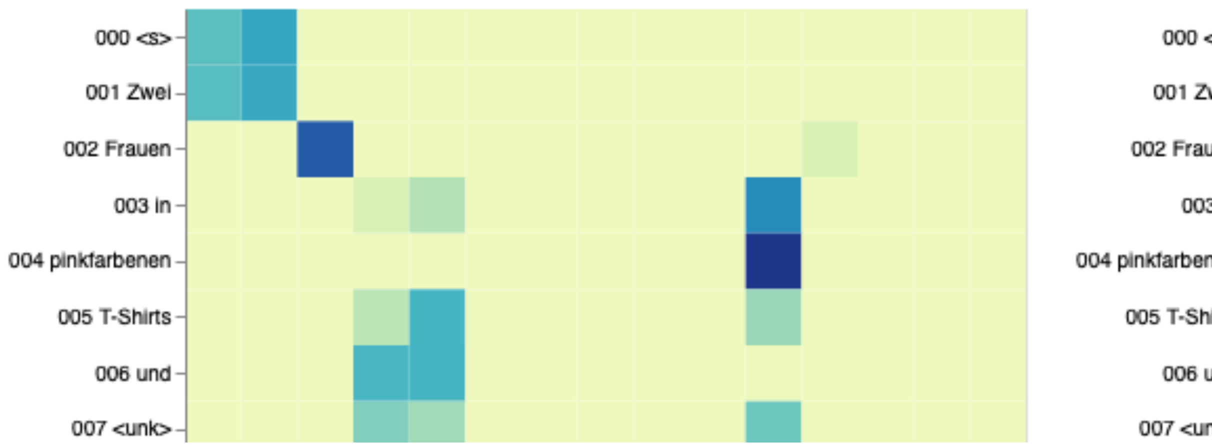
```python
def viz_encoder_self():
    model, example_data = run_model_example(n_examples=1)
    example = example_data[
        len(example_data) - 1
    ]  # batch object for the final example

    layer_viz = [
        visualize_layer(
```

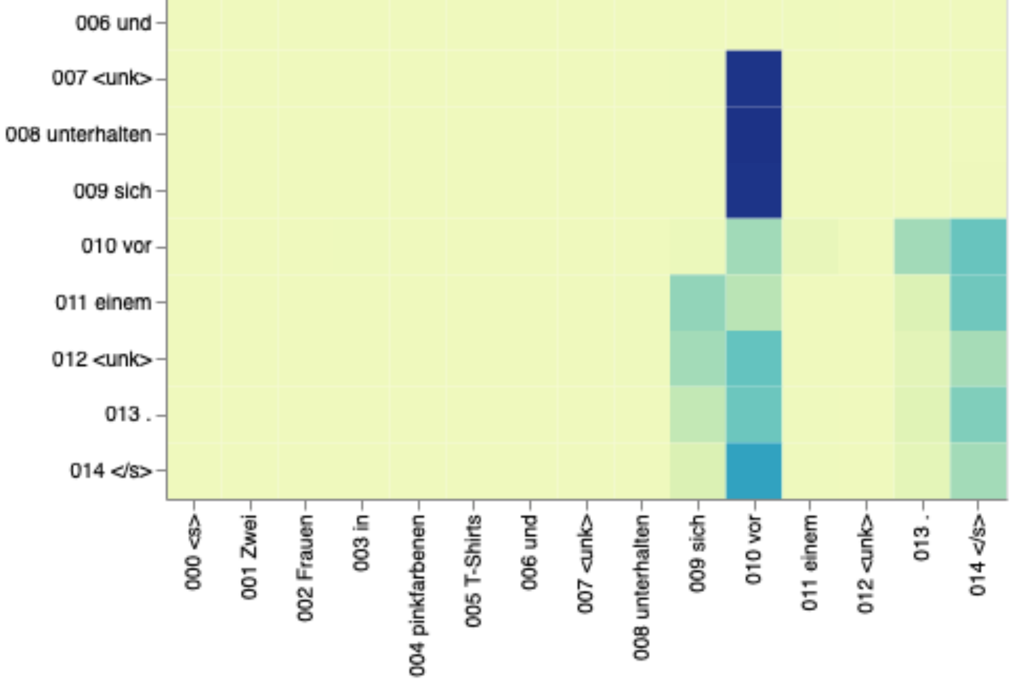
```
Example 0 ========

Source Text (Input)        : <s> Zwei Frauen in pinkfarbenen T-S
unterhalten sich vor einem <unk> . </s>
Target Text (Ground Truth) : <s> Two women wearing pink T - shir
converse outside clothing store . </s>
Model Output               : <s> Two women in pink shirts and fa
of a <unk> . </s>
```

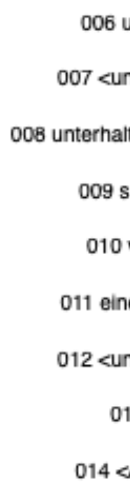
Layer 1



Layer 3

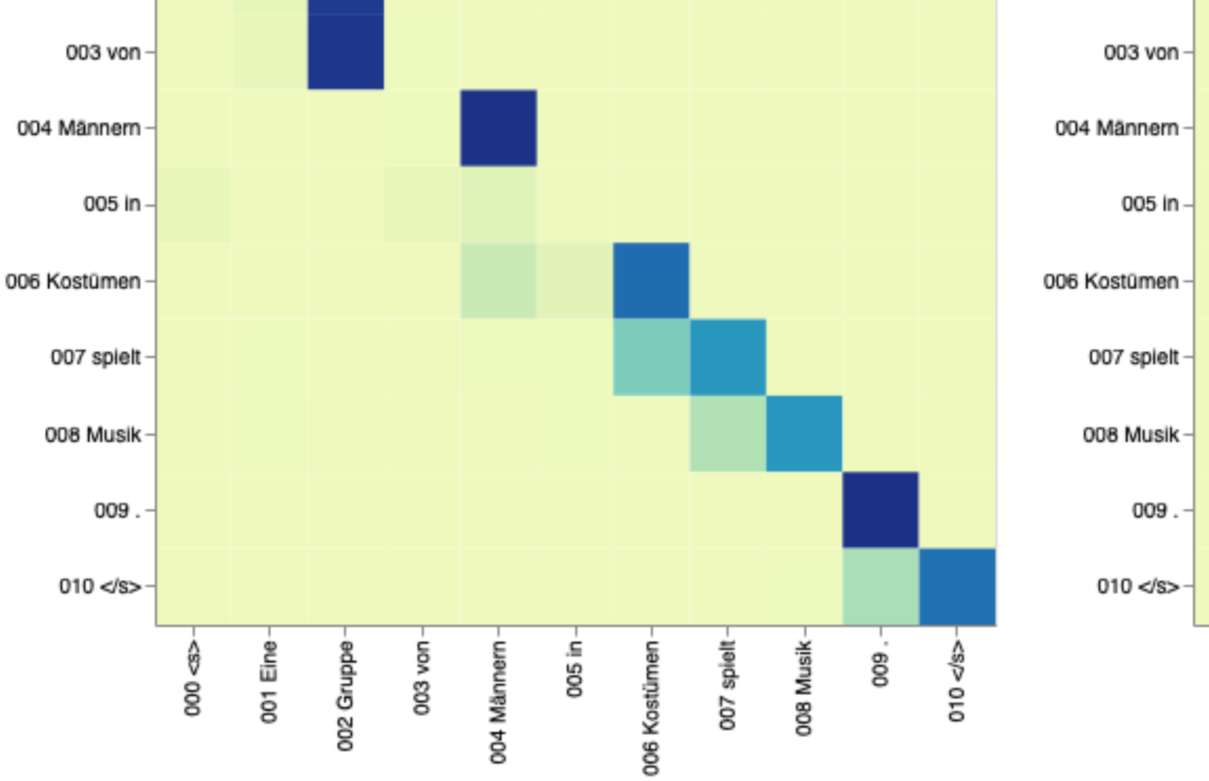




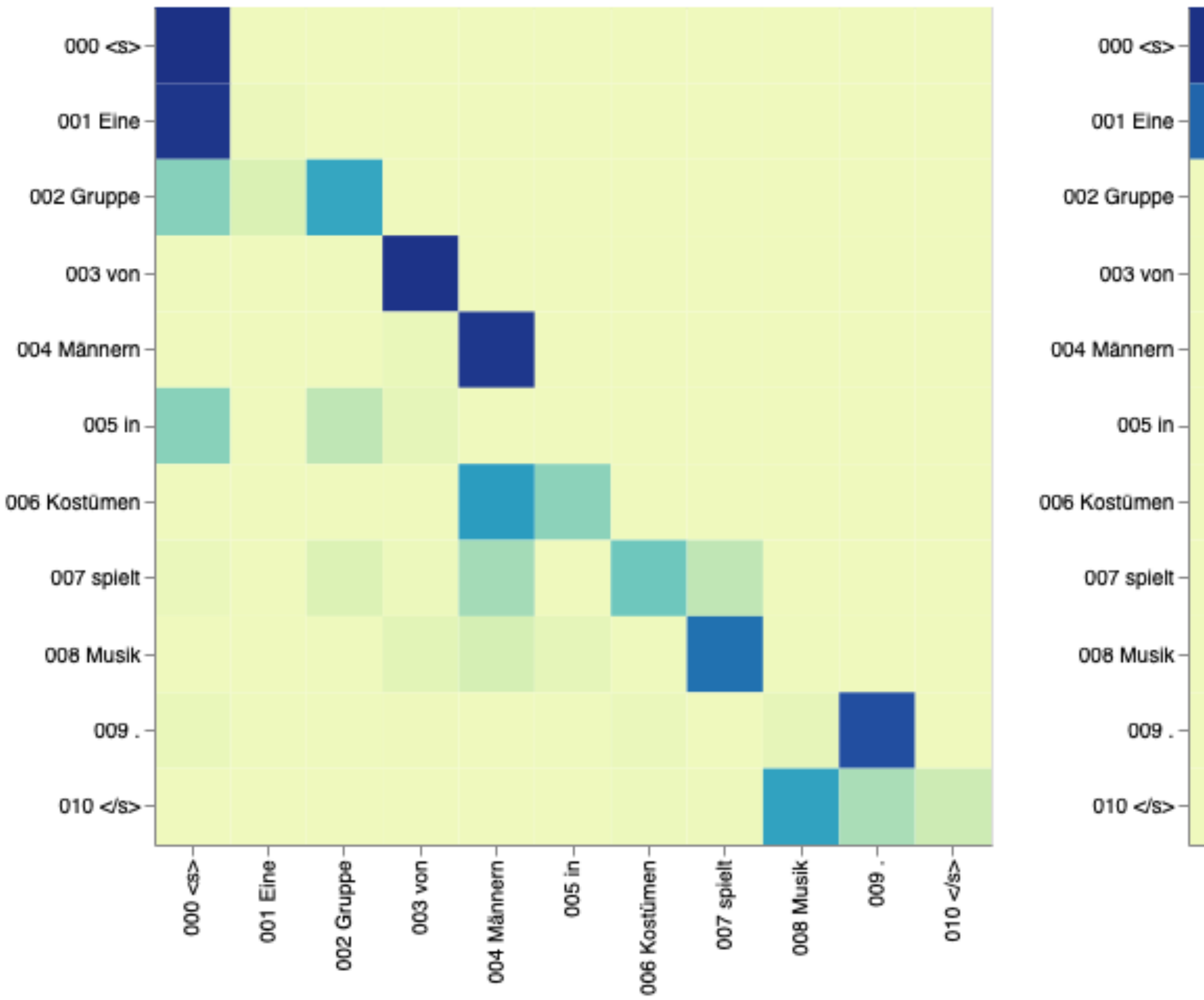


Decoder Self Attention

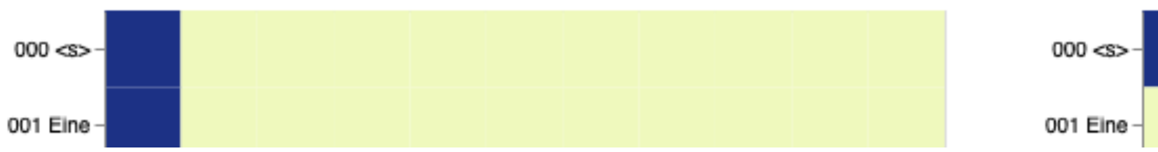
```
def viz_decoder_self():
    model, example_data = run_model_example(n_examples=1)
    example = example_data[len(example_data) - 1]

    layer_viz = [
        visualize_layer(
            model,
            layer,
            get_decoder_self,
            len(example[1]),
            example[1],
            example[1],
        )
        for layer in range(6)
    ]
    return alt.hconcat(
        layer_viz[0]
        & layer_viz[1]
        & layer_viz[2]
        & layer_viz[3]
        & layer_viz[4]
        & layer_viz[5]
    )
```

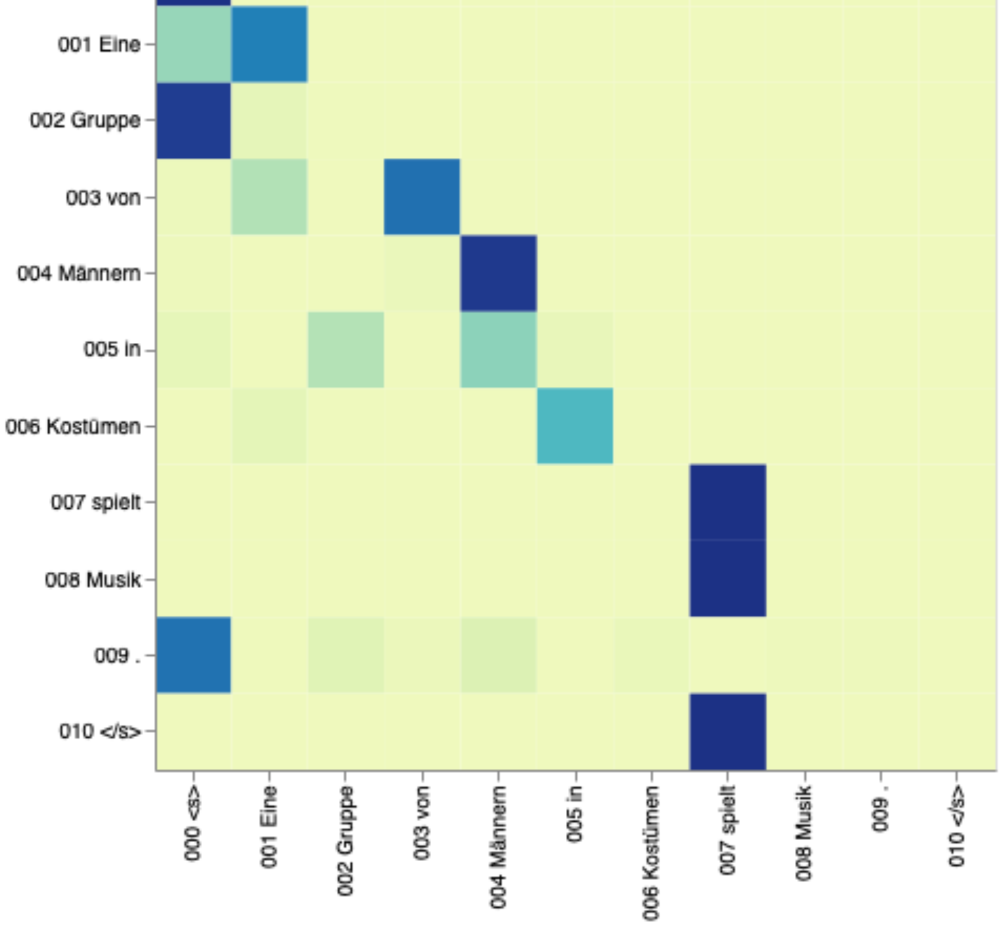


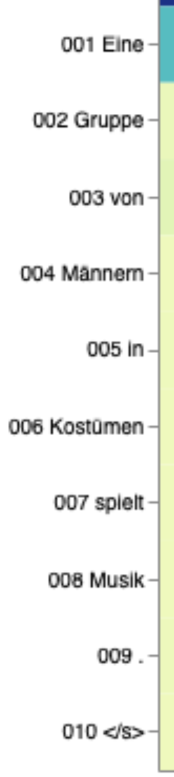
Layer 2



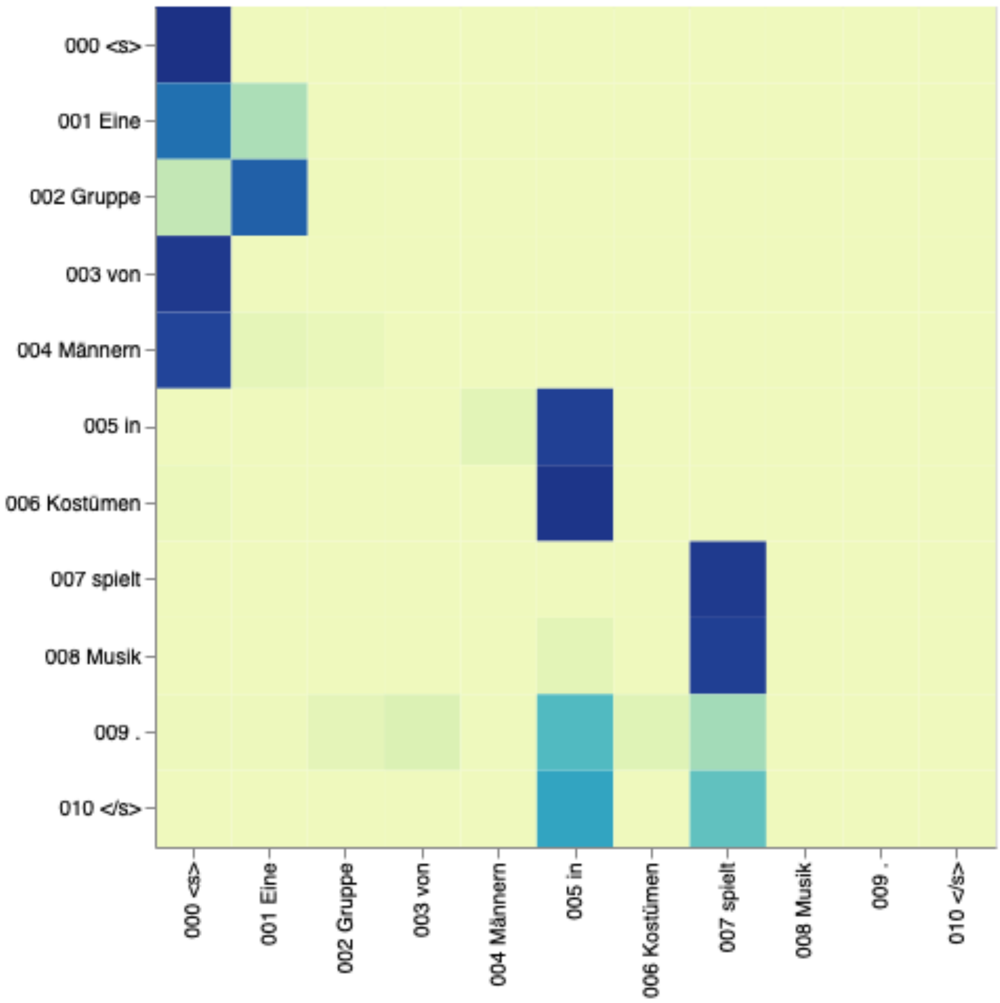
Layer 3

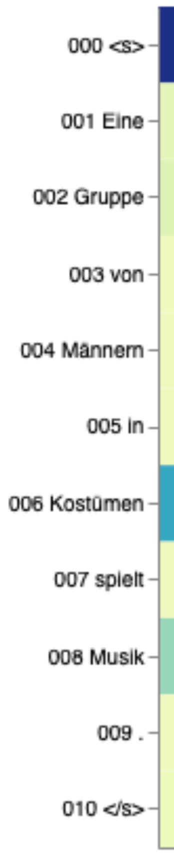






Layer 5



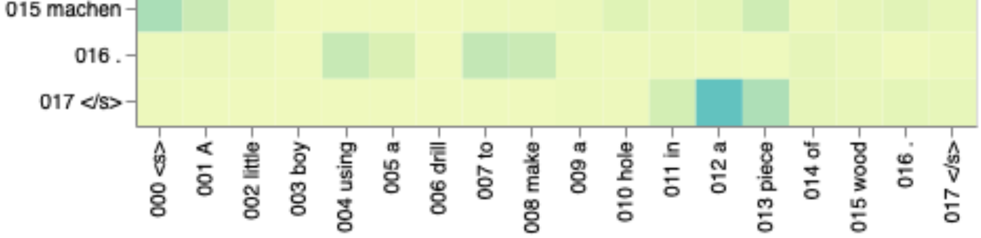


Layer 6

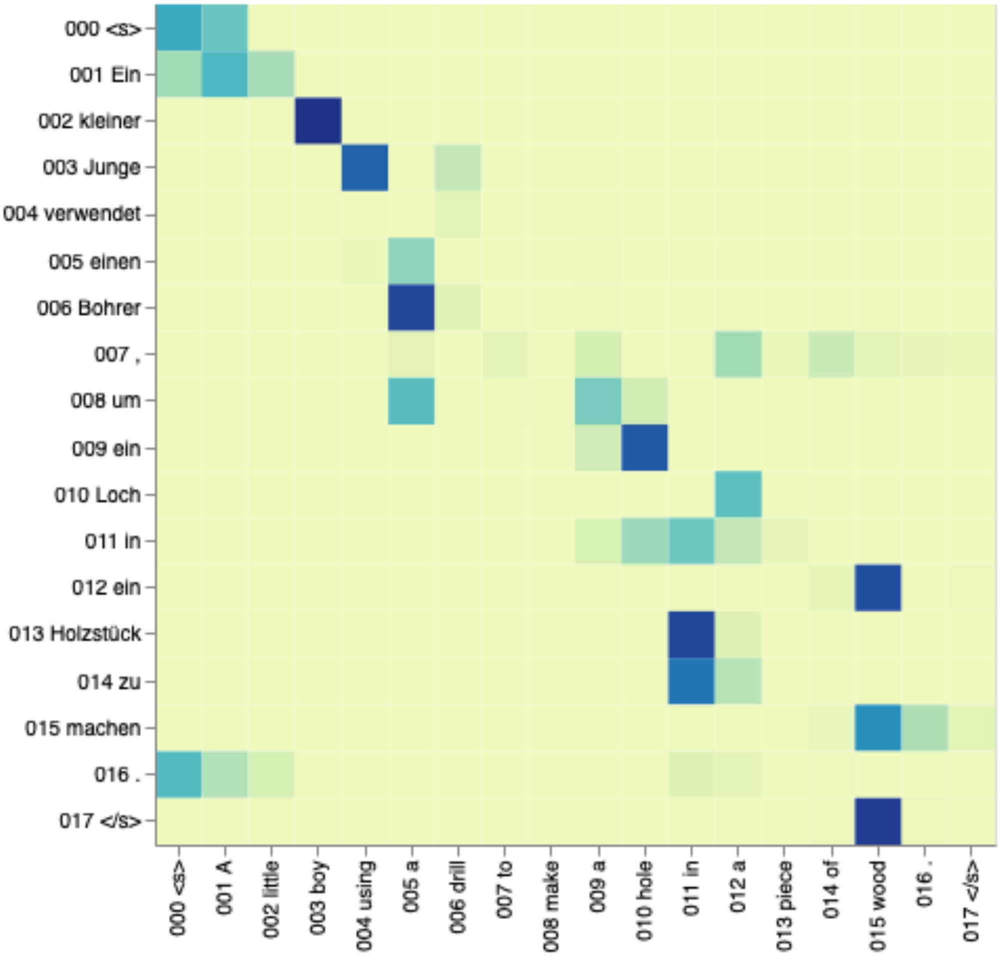
Decoder Src Attention

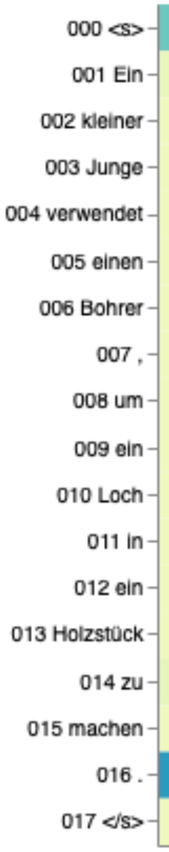
```python
def viz_decoder_src():
    model, example_data = run_model_example(n_examples=1)
    example = example_data[len(example_data) - 1]

    layer_viz = [
        visualize_layer(
            model,
            layer,
            get_decoder_src,
            max(len(example[1]), len(example[2])),
            example[1],
            example[2],
        )
        for layer in range(6)
    ]
    return alt.hconcat(
        layer_viz[0]
        & layer_viz[1]
        & layer_viz[2]
        & layer_viz[3]
        & layer_viz[4]
        & layer_viz[5]
    )


show_example(viz_decoder_src)
```

```
Preparing Data ...
Loading Trained Model ...
Checking Model Outputs:

Example 0 ========

Source Text (Input)        : <s> Ein kleiner Junge verwendet ein
in ein Holzstück zu machen . </s>
```

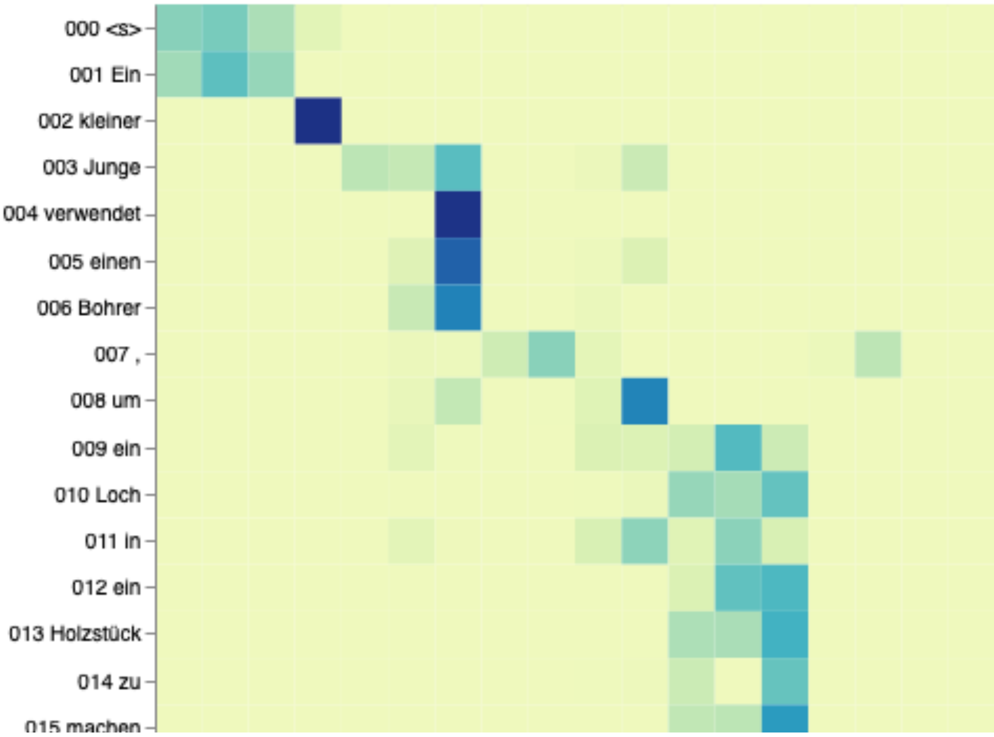


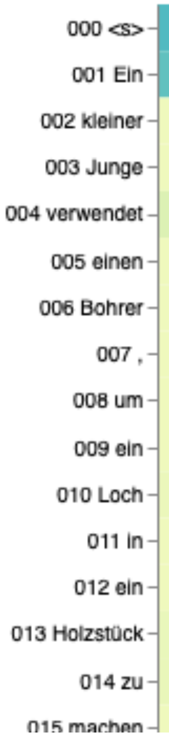
Layer 2

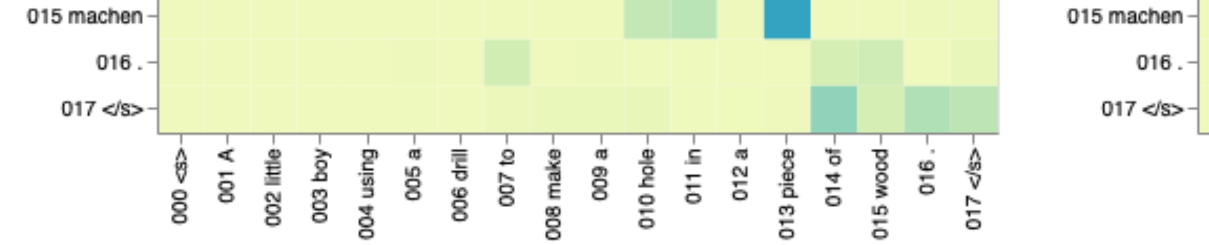




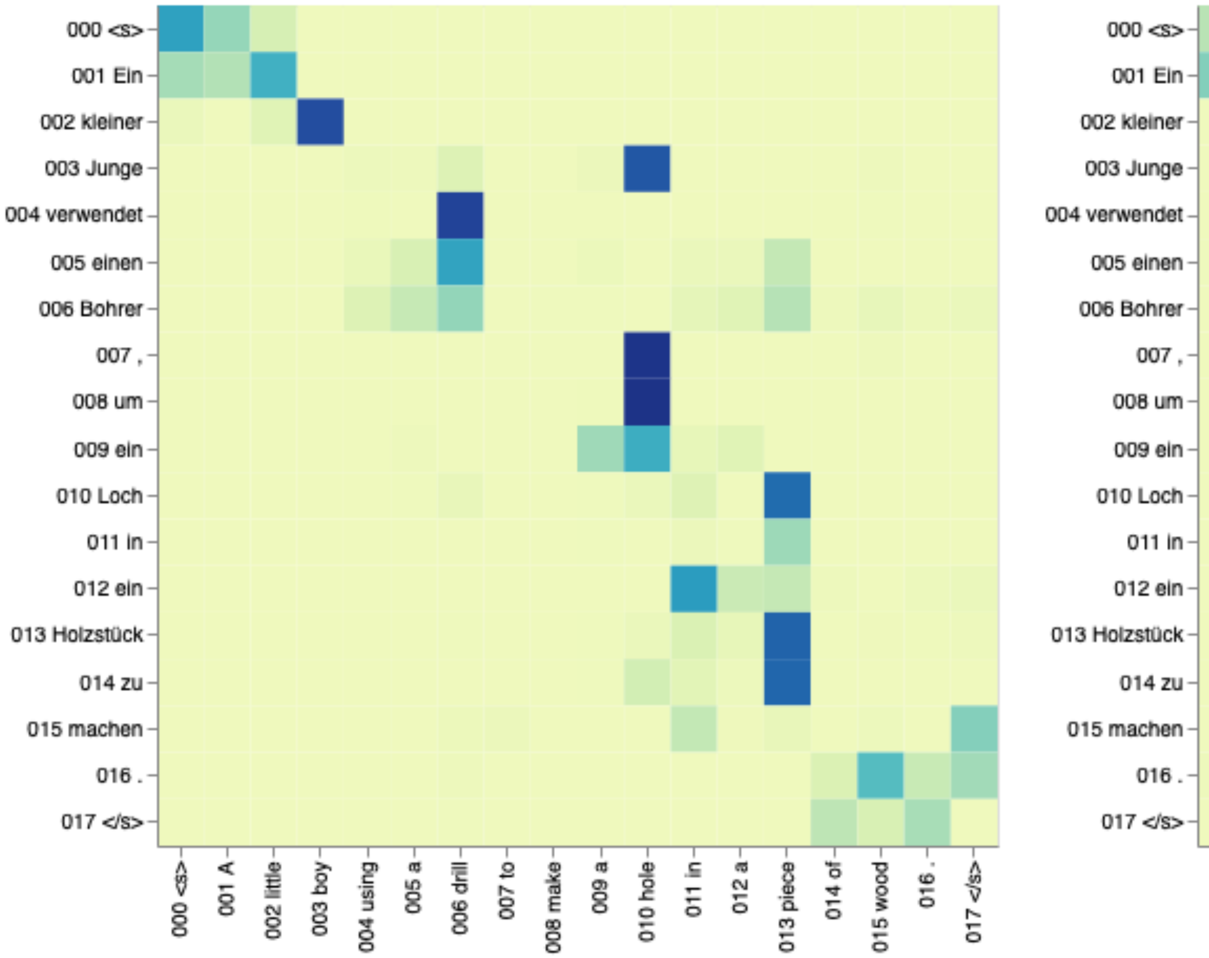
Layer 3







Layer 5



Layer 6