HW3: Neural Machine Translation

Jiafeng Chen Yufeng Ling Francisco Rivera

March 7, 2019

1 Introduction

In this writeup we consider neural machine translation—given a source sentence in one language, can we generate a target sentence in another? We work with the prevailing encoder-decoder architecture, where one neural network, the *encoder*, maps the source sentence into a numerical tensor—which is supposed to be a latent representation of the source sentence. Another network, the *decoder*, is a language model that takes the encoded sentence as an input, and generates a sentence in the target language.

2 Problem Description

In this writeup, we consider the problem of machine translation. Let

$$\boldsymbol{x}_i = [x_{i1}, \ldots, x_{iS}] \tag{1}$$

be a source sentence, where each word belongs to some source vocabulary, $x_{it} \in \mathcal{V}_s$. Let

$$\boldsymbol{y}_i = [y_{i1}, \ldots, y_{iT}] \tag{2}$$

be a target sentence, with each target word belonging to some target vocabulary, $y_{it} \in \mathcal{V}_t$. Our goal is to learn $p(\boldsymbol{y}_i \mid \boldsymbol{x}_i)$. We often treat the prediction like a language model—i.e. we consider the sequential conditional distributions $p(y_{it} \mid y_{i1}, \ldots, y_{i,t-1}, \boldsymbol{x}_i)$.

3 Model and Algorithms

3.1 Sequence-to-Sequence (Seq2Seq)

In Seq2Seq (Sutskever et al. (2014)), we have a encoder-decoder network architecture. The *encoder*, in the vanilla Seq2Seq implementation, is an LSTM network that takes an embedding of the source sentence $\mathsf{embed}(\boldsymbol{x}_{i1}), \ldots, \mathsf{embed}(\boldsymbol{x}_{iS})$ and outputs a list of hidden states $\boldsymbol{h}_{i1}, \ldots, \boldsymbol{h}_{iS}$ and cell state $\boldsymbol{c}_{iS}$.

In Seq2Seq, the decoder is another LSTM which is initialized with $\boldsymbol{h}_{iS}$ and $\boldsymbol{c}_{iS}$. At training time, the decoder LSTM takes embeddings of the ground truth target $\mathsf{embed}(y_{it})$ and output probability predictions for $y_{i,t+1}$. These predictions are penalized with the usual cross entropy loss at training time. At prediction time, the decoder LSTM gets passed the start-of-sentence token <s> and outputs predictions for the first word. We then iteratively pass in its top predictions to obtain predictions for future words. For instance, a greedy algorithm to generate a sentence would be taking the top prediction every time and pass the predicted sentence so far into the decoder to obtain the next word—stopping when the end-of-sentence token </s> is the top prediction.[1]

3.2 Attention

The attention model (Vaswani et al. (2017)) builds upon the Sequence-to-Sequence model described in section (3.1). Instead of only using the end state of the encoded input as input, we take advantage of a bi-directional RNN and generate target sentence by taking in a weighted input of all the states of the encoder. The intuition is that when we are translating a sentence, the adjacent words of the current word also help inform how we should translate the current word. Such weighted average is aptly named "context". To formalize this, we let the sequential conditional distribution be given by

$$p(y_{it} \mid y_{i1}, \ldots, y_{i,t-1}, \boldsymbol{x}_i) = g(y_{i,t-1}, s_t, c_t) \tag{3}$$

[1]This corresponds to beam search with beam size 1.

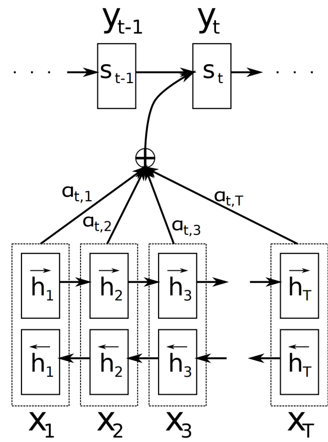

Figure 1: Attention Diagram

where s_t is the hidden state from BiRNN at time t defined by

$$s_t = f(s_{t-1}, y_{i,t-1}, c_t). \tag{4}$$

As mentioned above, the context vector c_t is a weighted average given by

$$c_t = \sum_{j=1}^{T_x} \alpha_{tj} h_j \tag{5}$$

where the annotations $\{h_j\}$ are the concatenated hidden states of the BiRNN and $\{\alpha_{tj}\}$ are the weights, which usually center around the current state t. The weights

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T_x} \exp(e_t k)} \tag{6}$$

are generated by softmaxing over

$$e_{tk} = a(s_{t-1}, h_k) \tag{7}$$

for some function a. In our model, we take a to be the dot product between the two arguments.

3.3 Beam Search

After we have trained a model, since the outputs are conditional probabilities over the entire vocabulary, we need to have an algorithm that generates the top k most likely sentences. One can use the greedy algorithm by selecting the local MLE conditional on the previously selected words. However, this can easily run into problems like stucking at a local optimum.

Beam search is proposed based on this idea. Instead of selecting the MLE, we select the top B words with the largest conditional probabilities. The parameter B is known as the "beam width". At every step, we choose the top B words with the highest joint probability conditional on the B previously generated sentenses $\{\boldsymbol{y}_{i,1:t-1}^{(b)}\}_{b=1}^{B}$. That is

$$\arg\max_{y_{it},b} p(y_{it}, y_{i,t-1}^{(b)}, \dots, y_{i1}^{(b)}, \boldsymbol{x}_i) = \arg\max_{y_{it},b} p(y_{it} \mid y_{i,t-1}^{(b)}, \dots, y_{i1}^{(b)}, \boldsymbol{x}_i) p(y_{i,t-1}^{(b)}, \dots, y_{i1}^{(b)}, \boldsymbol{x}_i).$$

In the last step we choose the top k to report as results.

3.4 Transformer extension

We also implemented the *transformer* architecture, following Vaswani et al. (2017). Each transformer layer is a self-attention layer,[2] a source-attention layer (only in the decoder), and a fully-connected feedforward network, connected by residual connections[3] and layer normalization.[4] We transform the input via a dense embedding and a *positional embedding*—treating position in a sentence as tokens in a vocabulary, and map to a dense embedding. We concatenate the embeddings and feed into the transformer layers. The model architecture is diagramed in Figure 2.

We trained two transformer models, which we name `Megatron` and `Soundwave`, after beloved *Transformers* antagonists. The `size` parameter is the number of dimensions of the dense embeddings, input to layers, and output from layers (so

[2]The attention layers are *multiheaded*, which means that we transform the (query, key, value) inputs by a linear layer before using dot-product attention (and normalization by $\sqrt{d}$. We then concatenate the resulting context vectors and feed into one last linear layer).

[3]A residual connection of a layer adds the input to the layer output. This helps with gradient propagation

[4]Layer normalization standardizes a layer's output along dimension of the dense embedding (i.e. the dimension other than the batch and seqlen dimensions).

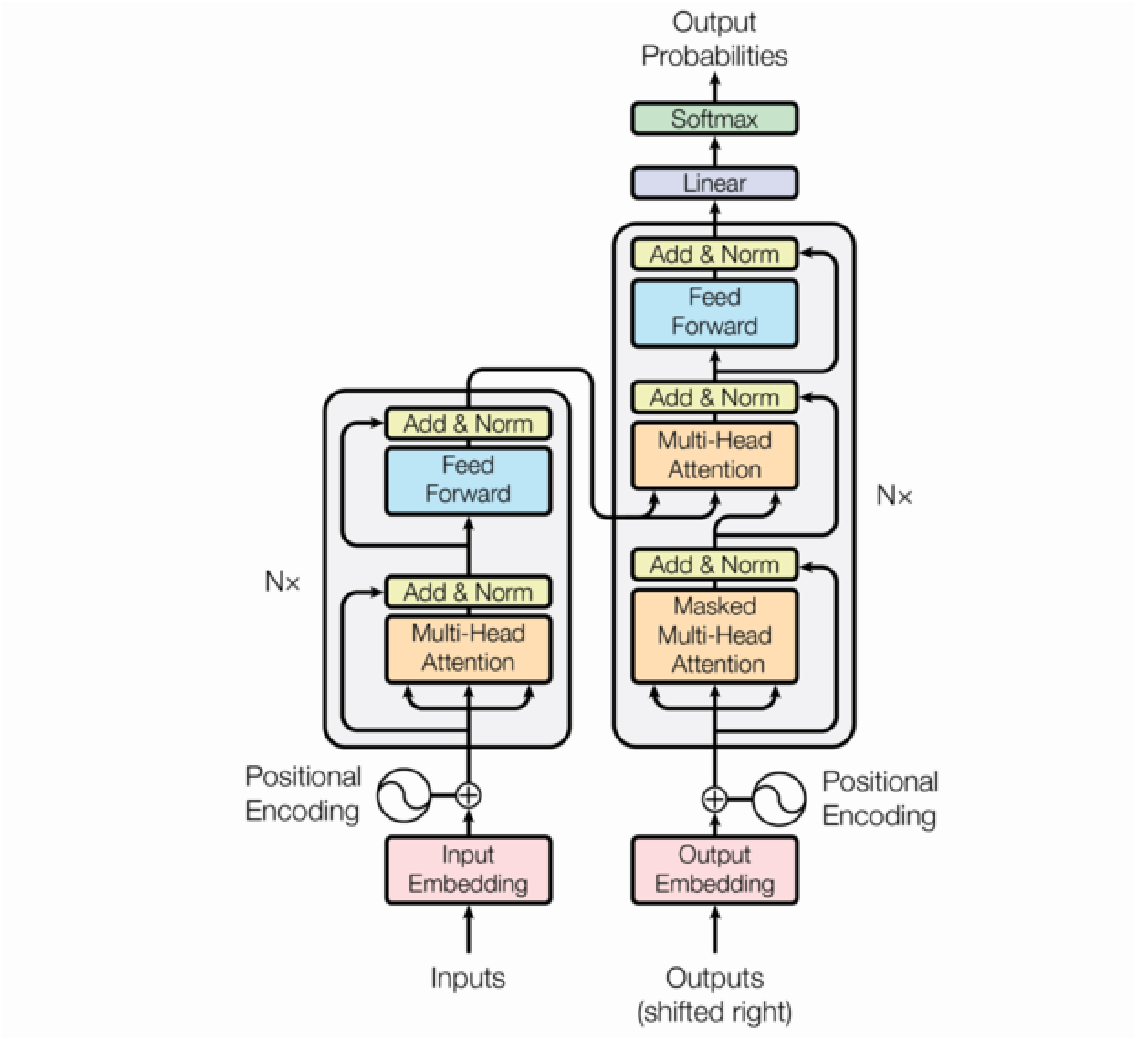

Figure 2: Diagram in Vaswani et al. (2017)

that the layers plug into each other). The `pos`, `emb` parameters are the number of dimensions in the positional and the word embedding

```
# Megatron
size = 200
pos = 20
emb = 180
head = 4
nlayers = 3
dropout = .2
```

```
# Soundwave
size = 200
pos = 20
emb = 180
head = 3
nlayers = 4
dropout = .1
```

4 Experiments

We trained two models and document hyperparameters in Table 1. Results are displayed in Table 2. The performance of the Sequence-to-Sequence model in Sutskever et al. (2014) is reasonable, but understandabely weaker due to its simple structure of combining two LSTMs without any attention layer applied to the source sentence.

In the attention model proposed in Bahdanau et al. (2014), we used 1/5 of the number of parameters compared to the paper due to the constraint of training speed. The main difference is that instead of a 5-layer decoder, we used one layer. Compared to the previous Sequence-to-Sequence model without attention, the performance improved significantly as seen in Table 2. We experimented normalizing the log-attention weights by factor of $\sqrt{\text{embedding size}}$ before proceeding to the softmax function to calculate the attention weights; this is because with more dimensions, dot product tends to be larger, and the softmax tends to be sharper, which would result in small gradients. However, this resulted in uniform weights across all source words and very poor performance. In the properly trained model, we can observe the attention weights reflecting our intuition. See Figure 3 and 4 for visualization of the attention weights.

It is also somewhat surprising to us that transformer model in Vaswani et al. (2017) does not perform well relative to LSTM-based models, contradicting established wisdom. We suspect that the reason is due to either poor implementation/optimization or short sentences and insufficient computing power. Attention is powerful when sentences are long, since it allows the model to look at arbitrarily long dependencies, and attention scales well with GPU hardware

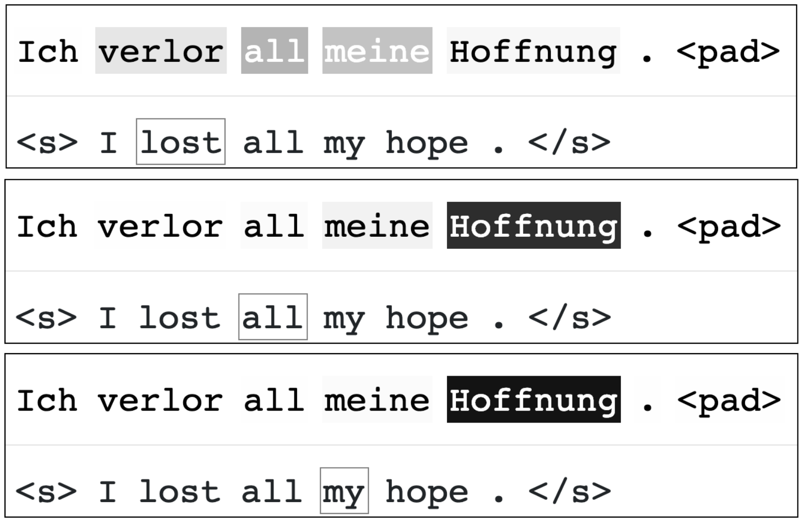

Figure 3: Darker background words signify more attention being placed on that word at each point in the translation process.

and multi-GPU training—we would get neither of these advantages with a small dataset and Google Colab computing power (Training `Megatron` takes 15 minutes per epoch and training `Soundwave` takes over 20 minutes, which does not leave room for hyperparameter tuning).

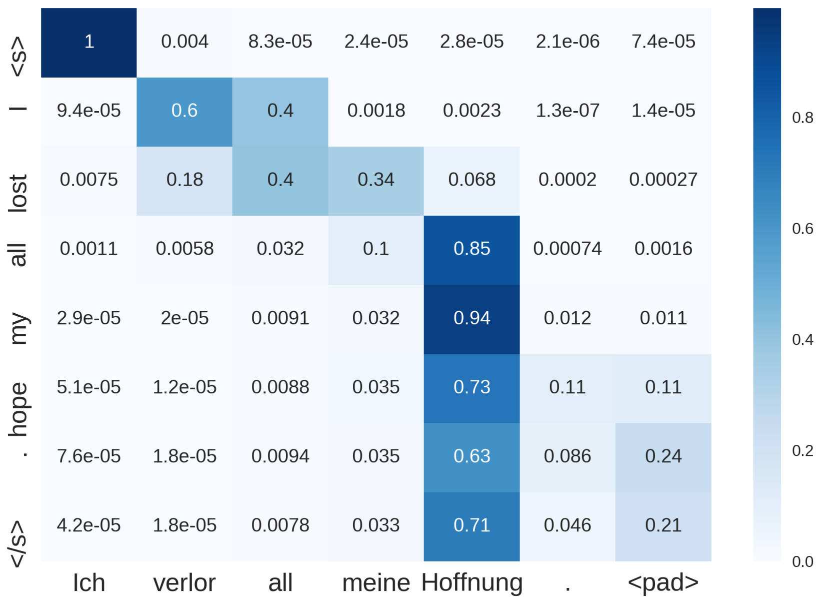

Figure 4: Attention matrix weights

model name	specifications
`Seq2Seq`	100 embedding, 100 hidden, no dropout
`Seq2SeqAttn`	Bi-directional LSTM, 100 embedding, 100 hidden, no dropout

Table 1: Both models used only one layer in the decoder and trained with Adam and learning rate 10^{-3} *over 10 epochs. For* `Seq2Seq`*, decrease in validation loss flattened at epoch 8. For* `Seq2SeqAttn`*, training stopped at epoch 7 after loss starting going up.*

	Loss	Perplexity	BLEU
model name			
`Seq2Seq`	3.23	25.36	6.32
`Seq2SeqAttn`	2.71	15.05	17.77
`Megatron`	2.7	15	9.68
`Soundwave`	2.8	16	7.08

Table 2: Performance metrics for different models

References

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

A Model implementation

Listing 1: Sequence-to-Sequence

```
from namedtensor import ntorch
from namedtensor.nn import nn as nnn

class Seq2Seq(nnn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
    def forward(self, src, trg):
        encoded = self.encoder(src)
        return self.decoder(encoded, trg)


ce_loss = nnn.CrossEntropyLoss().spec("classes")

# def seq2seq_loss_fn(model, batch):
# length = batch.trg.size('trgSeqlen')
# out = model(batch.src, batch.trg)[{'trgSeqlen': slice(0, length - 1)}]
# trg = batch.trg[{'trgSeqlen': slice(1, length)}]
# mask = (trg != pad_idx_EN).float()

# n = trg.values.numel()
# loss = ce_loss(mask * out, trg)

# return (loss * n - np.log(V_EN) * (n - mask.sum().item())) / mask.sum().item()
```

Listing 2: LSTM encoder

```
from namedtensor import ntorch
from namedtensor.nn import nn as nnn
```

```python

class LSTMEncoder(nnn.Module):
    def __init__(self, TEXT,
                 embedding_dim=100,
                 hidden_dim=150,
                 num_layers=1,
                 dropout=0,
                 history=False,
                 bidirectional=False):
        super().__init__()
        self.history = history
        pad_idx = TEXT.vocab.stoi['<pad>']

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embedding_dim,
                                   padding_idx=pad_idx)

        self.lstm = nnn.LSTM(input_size=embedding_dim,
                             hidden_size=hidden_dim,
                             num_layers=num_layers,
                             dropout=dropout,
                             bidirectional=bidirectional) \
                        .spec("embedding", "srcSeqlen")

    def forward(self, batch_text):
        embedded = self.embed(batch_text)
        hidden_states, last_state = self.lstm(embedded)
        if self.history:
            return hidden_states, last_state
        else:
            return last_state
```

Listing 3: LSTM decoder

```python
import torch
from torch import nn
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn
from attention import Attention
from utils import *

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')


class LSTMDecoder(nnn.Module):
    def __init__(self, TEXT,
                 embedding_dim=100,
                 hidden_dim=150,
                 num_layers=1,
                 dropout=0):
        super().__init__()

        pad_idx = TEXT.vocab.stoi['<pad>']

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embedding_dim,
                                   padding_idx=pad_idx)

        self.lstm = nnn.LSTM(input_size=embedding_dim,
```

```
                          hidden_size=hidden_dim,
                          num_layers=num_layers,
                          dropout=dropout) \
            .spec("embedding", "trgSeqlen")

        self.w = nnn.Linear(in_features=hidden_dim,
                            out_features=len(TEXT.vocab)) \
            .spec("embedding", "classes")

    def forward(self, init_state, batch_text):
        embedded = self.embed(batch_text)
        hidden_states, (hn, cn) = self.lstm(embedded, init_state)
        log_probs = self.w(hidden_states)
        return log_probs


class LSTMDecoderAttn(nnn.Module):
    def __init__(self, TEXT,
                 embedding_dim=100,
                 hidden_dim=150,
                 num_layers=1,
                 dropout=0,
                 attn_normalize=True):
        super().__init__()

        pad_idx = TEXT.vocab.stoi['<pad>']

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embedding_dim,
                                   padding_idx=pad_idx)

        self.lstm = nnn.LSTM(input_size=embedding_dim + hidden_dim,
                          hidden_size=hidden_dim,
                          num_layers=num_layers,
                          dropout=dropout) \
            .spec("embedding", "trgSeqlen")

        self.w = nnn.Linear(in_features=hidden_dim,
                            out_features=len(TEXT.vocab)) \
            .spec("embedding", "classes")

        self.attention = []

        self.attn = Attention(attn_normalize)

    def forward(self, init_state, batch_text):
        H, (ht, ct) = init_state
        ht, ct = (unsqueeze(flatten(ht, "layers", "embedding"), "layers"),
                  unsqueeze(flatten(ct, "layers", "embedding"), "layers"))

        embedded = self.embed(batch_text)

        hidden_states = []
        attention_weights = []
        ht_flat = flatten(ht, "layers", "embedding")

        for t in range(embedded.shape["trgSeqlen"]):

            a, context = self.attn(ht_flat, H, "embedding", "srcSeqlen")
```

```python
            context = unsqueeze(context, "trgSeqlen")
            word_t = embedded[{'trgSeqlen': slice(t, t+1)}]
            lstm_input = ntorch.cat([word_t, context], "embedding")
            output, (ht, ct) = self.lstm(lstm_input, (ht, ct))
            if not self.training:
                attention_weights.append(a)
            ht_flat = flatten(ht, "layers", "embedding")
            hidden_states.append(ht_flat)

        if not self.training:
            self.attention = ntorch.stack(attention_weights, 'trgSeqlen')
        return self.w(ntorch.stack(hidden_states, "trgSeqlen"))
```

Listing 4: Attention

```python
import torch
from torch import nn
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

class Attention(nnn.Module):
    def __init__(self, normalize=True):
        super().__init__()
        self.normalize = normalize

    def forward(self, query, key, embedding_dim, softmax_dim,
                mask=None, value=None, dropout=None,
                self_attention=False):
        if self.normalize:
            log_weights = (query.dot(embedding_dim, key)
                           / (key.size(embedding_dim) ** .5))
        else:
            log_weights = query.dot(embedding_dim, key)

        if mask is not None:
            log_weights = log_weights.masked_fill_(mask == 0, -1e9)

        if value is None:
            value = key

        a = log_weights.softmax(softmax_dim)

        if dropout is not None:
            a = dropout(a)

        return (a, (a * value).sum(softmax_dim))
```

Listing 5: Beam search

```python
import torch
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

class Beam(nnn.Module):
```

```
    def __init__(self, TEXT, beam_size=3, topk=10, max_len=50):
        super().__init__()
        self.TEXT = TEXT
        self.beam_size = beam_size
        self.topk = topk
        self.max_len = max_len
        self.scores = None
        self.nodes = None
        self.result = None
        self.filled = None
        self.top_scores = None
        self.top_score_locs = None

    def log_prob(self, model, src):
        # calculate marginal probability for next word
        if self.nodes.shape['trgSeqlen'] == 1:
            # start with SOS
            log_prob = model(src, self.nodes[{'beam': 0}])[{'trgSeqlen': -1}]
            return log_prob, log_prob.shape['classes']
        else:
            log_prob = [model(src, self.nodes[{'beam': b}])[{'trgSeqlen': -1}]
                        + self.scores[{'beam': b}]
                        for b in range(self.beam_size)]
            return ntorch.cat(log_prob,'classes'), log_prob[0].shape['classes']

    def generate_sentence(self, b, k, vocab_size):
        sentence_i = self.top_score_locs[{'batch': b, 'classes': k}] / vocab_size
        prev_sentence = self.nodes[{'batch': b, 'beam': sentence_i}].values[:-1]
        word = self.top_score_locs[{'batch': b, 'classes': k}].fmod(vocab_size)
        word = torch.LongTensor([word.values]).to(device)
        sentence = NamedTensor(torch.cat([prev_sentence, word]), names=('trgSeqlen'))

        return sentence

    def advance(self, batch_size, vocab_size):
        # increment length and pad using 1
        increment = ntorch.ones((batch_size, self.beam_size, 1),
                                names=('batch', 'beam', 'trgSeqlen')) \
                    .long().to(device)
        self.nodes = ntorch.cat([self.nodes, increment], 'trgSeqlen')

        # generate new next word for the entire batch
        for b in range(batch_size):
            beam_count = 0
            new_nodes = ntorch.zeros((self.nodes.shape['trgSeqlen'],
                                      self.beam_size),
                                     names=('trgSeqlen', 'beam')) \
                        .long().to(device)
            if len(self.result[b]) < self.topk:
                for k in range(self.topk):
                    sentence = self.generate_sentence(b, k, vocab_size)
                    added_word = sentence[{'trgSeqlen': -1}]
                    if len(self.result[b]) < self.topk and \
                        (added_word.values.item() == self.TEXT.vocab.stoi["</s>"]
                         or self.nodes.shape['trgSeqlen'] == self.max_len + 1):
                        self.result[b].append(sentence)
                    else:
                        new_nodes[{'beam': beam_count}] = sentence
                        self.scores[{'batch': b, 'beam': beam_count}] = \
```

```
                            self.top_scores[{'batch': b, 'classes': k}].values.item()
                        beam_count += 1
                        if beam_count == self.beam_size:
                            break
                self.nodes[{'batch': b}] = new_nodes
            else:
                self.filled[b] = True

    def forward(self, model, src):
        batch_size = src.shape['batch']
        self.nodes = (ntorch.ones((batch_size, self.beam_size, 1),
                                  names=('batch', 'beam', 'trgSeqlen'))
                     * self.TEXT.vocab.stoi["<s>"]).long().to(device)
        self.scores = ntorch.zeros((batch_size, self.beam_size),
                                   names=('batch', 'beam')).to(device)
        self.result = [[] for _ in range(batch_size)]
        self.filled = [False] * batch_size

        while (sum(self.filled) < batch_size and
            self.nodes.shape['trgSeqlen'] <= self.max_len):
            log_prob, vocab_size = self.log_prob(model, src)
            self.top_scores, self.top_score_locs = log_prob.topk('classes', self.topk)
            self.advance(batch_size, vocab_size)

        return self.result
```

A.1 Transformer implementation

Listing 6: Encoder Layer

```
from namedtensor.nn import nn as nnn
from sublayer_connection import SublayerConnection

class TransformerEncoderLayer(nnn.Module):
    def __init__(self, attn, feed_forward, size, dropout=.3):
        super().__init__()
        self.attn = attn
        self.feed_forward = feed_forward

        self.sublayer = nnn.ModuleList([
                SublayerConnection(size, dropout, "embedding")
                for _ in range(2)])
        self.size = size


    def forward(self, src):
        a, x = self.sublayer[0](src, lambda src: self.attn(src, src, "embedding",
            "srcSeqlen"))
        return self.sublayer[1](x, self.feed_forward)
```

Listing 7: Decoder Layer

```
from namedtensor.nn import nn as nnn
from sublayer_connection import SublayerConnection
from namedtensor import ntorch, NamedTensor


```

```python
class TransformerDecoderLayer(nnn.Module):
    def __init__(self, self_attn, src_attn, feed_forward, size, dropout=.3):
        super().__init__()
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward

        self.sublayer = nnn.ModuleList([
                SublayerConnection(size, dropout, "embedding")
                for _ in range(3)])
        self.size = size


    def forward(self, encoded, trg):
        trglen = trg.size("trgSeqlen")
        mask = ntorch.triu(ntorch.ones(trglen, trglen, names=('src','trg'))
                          .to(trg.values.device),
                          dims=("src","trg"))
        a, x = self.sublayer[0](trg, lambda x: self.self_attn(x, x, "embedding", "trgSeqlen", mask=mask))
        a, x = self.sublayer[1](trg, lambda x: self.src_attn(x, encoded, "embedding", "srcSeqlen"))

        return a, self.sublayer[2](x, self.feed_forward)
```

Listing 8: Encoder

```python
from namedtensor.nn import nn as nnn
from copy import deepcopy
from layernorm import LayerNorm
from namedtensor import ntorch, NamedTensor

MAX_LEN = 20


class TransformerEncoder(nnn.Module):
    def __init__(self, embed_dim, position_dim, TEXT, layer, nlayers=3):
        super().__init__()
        size = layer.size
        pad_idx = TEXT.vocab.stoi['<pad>']
        assert embed_dim + position_dim == size, \
            "Embedding dimension + position_embedding must equal size"

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embed_dim,
                                   padding_idx=pad_idx)

        self.position_embed = nnn.Embedding(num_embeddings=MAX_LEN + 1,
                                   embedding_dim=position_dim,
                                   padding_idx=MAX_LEN)

        self.layers = nnn.ModuleList([deepcopy(layer) for _ in range(nlayers)])
        self.norm = LayerNorm(size, "embedding")

    def forward(self, x):
        pos = ntorch.ones(*x.shape.values(),
            names=[*x.shape.keys()]).to(x.values.device)
        pos_vec = ntorch.arange(x.size("srcSeqlen"), names="srcSeqlen").float()
        pos_vec[pos_vec > MAX_LEN] = MAX_LEN
```

```
        embed = self.embed(x)
        position_embed = self.position_embed((pos *
            pos_vec.to(x.values.device)).long())


        x = ntorch.cat([embed, position_embed], "embedding")
        for layer in self.layers:
            x = layer(x)
        return self.norm(x)
```

Listing 9: Decoder

```
from namedtensor.nn import nn as nnn
from copy import deepcopy
from layernorm import LayerNorm
from namedtensor import ntorch, NamedTensor
from position_encoding import PositionalEncoding

MAX_LEN = 20

class TransformerDecoder(nnn.Module):
    def __init__(self, embed_dim, position_dim, TEXT, layer, nlayers=3):
        super().__init__()
        size = layer.size
        pad_idx = TEXT.vocab.stoi['<pad>']
        assert embed_dim + position_dim == size, \
            "Embedding dimension + position_embedding must equal size"

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embed_dim,
                                   padding_idx=pad_idx)

        self.position_embed = nnn.Embedding(num_embeddings=MAX_LEN + 1,
                                   embedding_dim=position_dim,
                                   padding_idx=MAX_LEN)

        self.layers = nnn.ModuleList([deepcopy(layer) for _ in range(nlayers)])
        self.norm = LayerNorm(size, "embedding")
        self.w = nnn.Linear(in_features=size,
            out_features=len(TEXT.vocab)).spec("embedding", "classes")

    def forward(self, encoded, trg):
        # attn_weights = []
        pos = ntorch.ones(*trg.shape.values(),
            names=[*trg.shape.keys()]).to(trg.values.device)
        pos_vec = ntorch.arange(trg.size("trgSeqlen"), names="trgSeqlen").float()
        pos_vec[pos_vec > MAX_LEN] = MAX_LEN
        embed = self.embed(trg)
        position_embed = self.position_embed((pos *
            pos_vec.to(trg.values.device)).long())


        x = ntorch.cat([embed, position_embed], "embedding")
        for layer in self.layers:
            a, x = layer(encoded, x)
            # if not self.training:
            # attn_weights.append(a)
        # if not self.training:
        # self.attn_weights = ntorch.stack(self.attn_weights, "layers")
```

```
        x = self.norm(x)
        # self.attn_weights = attn_weights
        return self.w(x)
```

Listing 10: Sublayer connection

```
import torch
from torch import nn
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn
from attention import Attention
from layernorm import LayerNorm

class SublayerConnection(nnn.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, dim):
        super().__init__()
        self.norm = LayerNorm(size, dim)
        self.dropout = nnn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        out = sublayer(self.norm(x))
        if type(out) is tuple:
            a, context = out
            return a, x + self.dropout(context)
        else:
            return x + self.dropout(out)
```

Listing 11: Multihead Attention

```
import torch
from torch import nn
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn
from attention import Attention

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

class MultiHeadAttention(nnn.Module):
    """
    Takes query, key, value tuple and pass through n_head linear layers
    (with interim_dim_size output dimensions),
    attend, concatenate, and pass through one final linear layer.
    The layer maps input_dim_name of input_dim_size to
    input_dim_name with output_dim_size.
    """
    def __init__(self, n_heads, input_dim_name, input_dim_size,
                 interim_dim_size, output_dim_size, dropout=0, attn=None):
        super().__init__()
        self.linears = nnn.ModuleList([nnn.Linear(
            in_features=input_dim_size,
            out_features=interim_dim_size).spec(input_dim_name, input_dim_name)
        for _ in range(n_heads)])

```

```
        if attn is None:
            self.attn = Attention()
        else:
            self.attn = attn

        self.dropout = nnn.Dropout(dropout)
        self.linear_final = nnn.Linear(
            in_features=interim_dim_size * n_heads,
            out_features=output_dim_size).spec(input_dim_name, input_dim_name)

    def forward(self, query, key, embedding_dim, softmax_dim, mask=None, value=None):
        if value is None:
            value = key
        attended = [self.attn(l(query), l(key), embedding_dim,
            softmax_dim, mask=mask, value=l(value), dropout=self.dropout)
            for l in self.linears]
        a = [tup[0] for tup in attended]
        contexts = ntorch.cat([tup[1] for tup in attended], embedding_dim)
        return a, self.linear_final(contexts)
```

Listing 12: Self-attention

```
import torch
from torch import nn
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn
from attention import Attention

class SelfAttention(Attention):
    def __init__(self, *args):
        super().__init__(*args)
    def forward(self, query, key, embedding_dim, softmax_dim, mask=None,
        value=None, **kwargs):
        # Source is key,values, Target is query
        target_name = f"trg{softmax_dim}"
        trg = key.rename(softmax_dim, target_name)
        if mask is not None:
            mask = mask.rename("src", softmax_dim).rename("trg", target_name)

        a, context = super().forward(query, trg, embedding_dim, softmax_dim,
            mask, value=trg, **kwargs)

        return a, context.rename(target_name, softmax_dim)
```

Listing 13: Layer norm

```
import torch
from torch import nn
from namedtensor import ntorch, NamedTensor
from namedtensor.nn import nn as nnn

class LayerNorm(nnn.Module):
    def __init__(self, features, dim, eps=1e-6):
        super().__init__()
        self.a_2 = ntorch.ones(features, names=(dim,))
        self.b_2 = ntorch.ones(features, names=(dim,))
        self.register_parameter("layernorma", self.a_2)
        self.register_parameter("layernormb", self.b_2)
```

```
        self.eps = eps
        self.dim = dim

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