**Artificial Intelligence Final Report Assignment 問題3 (Problem 3)**

**レポート解答用紙 (Report Answer Sheet)**

**(Group 35)**

Group Leader

学生証番号 (Student ID): 21522357

名前(Name): Lê Hải Nam

Group Members

学生証番号 (Student ID): 21522429

名前(Name): Trần Nguyễn Yến Nhi

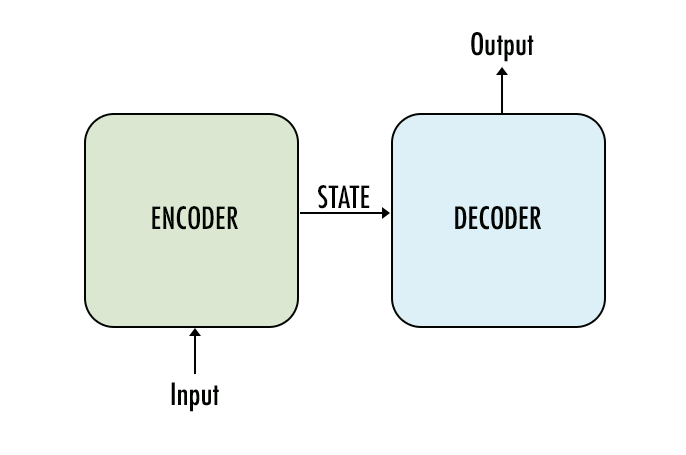
学生証番号 (Student ID): 21522315

名前(Name): Nguyễn Thị Cẩm Ly

問題3 (Problem 3)のレポート

**Idea:**

To improve the version of the program in the 13th lecture (or the program was written in Lab Work (6)). We do preprocess on the dataset and build a transformer model to increase BLEU. The best results in the 13th lecture with BLEU is about **2,5%** (LSTM with dropout).



*Figure 1: seq2seq model.*

**The seq2Seq model** is a kind of machine learning model that takes sequential data as input and generates sequential data as output. Before the arrival of Seq2Seq models, the machine translation systems relied on statistical methods and phrase-based approaches. The most popular approach was the use of phrase-based statistical machine translation (SMT) systems. That was not able to handle long-distance dependencies and capture global context.

**Program:**

**Prepare:**

**The fix\_str\_data function** performs preprocessing of the text string by replacing special and escaped characters with regular characters. We replace XHTML characters, which causes inaccuracies between English and Vietnamese.

Ex:

|  |  |  |
| --- | --- | --- |
|  | **English (**test\_en[428]) | **Vietnamese** (test\_vi[430]) |
| **Before** | I can&apos;t do it | &quot;Chạy ngay đi &quot; là một bài nhạc. |
| **After** | I can’t do it | “Chạy ngay đi” là một bài nhạc. |

Program:

def fix\_str\_data(string):

string = string.replace(" &apos;", "'")

string = string.replace("&quot;", '"')

string = string.replace("&#91;", '[')

string = string.replace("&#93;", ']')

string = string.replace("--", '-')

return string

1. **Define model**

This “Positional Encoding” module injects information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension as the embeddings so that they can be summed together.

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, emb\_size, dropout, maxlen = 5000):

super(PositionalEncoding, self).\_\_init\_\_()

den = torch.exp(- torch.arange(0, emb\_size, 2)\* math.log(10000) / emb\_size)

pos = torch.arange(0, maxlen).reshape(maxlen, 1)

pos\_embedding = torch.zeros((maxlen, emb\_size))

pos\_embedding[:, 0::2] = torch.sin(pos \* den)

pos\_embedding[:, 1::2] = torch.cos(pos \* den)

pos\_embedding = pos\_embedding.unsqueeze(-2)

self.dropout = nn.Dropout(dropout)

self.register\_buffer('pos\_embedding', pos\_embedding)

def forward(self, token\_embedding):

return self.dropout(token\_embedding + self.pos\_embedding[:token\_embedding.size(0), :])

class TokenEmbedding(nn.Module):

def \_\_init\_\_(self, vocab\_size, emb\_size):

super(TokenEmbedding, self).\_\_init\_\_()

self.embedding = nn.Embedding(vocab\_size, emb\_size)

self.emb\_size = emb\_size

def forward(self, tokens):

return self.embedding(tokens.long()) \* math.sqrt(self.emb\_size)

The Seq2Seq class implementation defines a sequence-to-sequence model architecture, leveraging a Transformer network for tasks such as language translation. This architecture includes an encoder-decoder structure, where both components are composed of multiple layers of self-attention and feed-forward neural networks. The encoder processes input sequences to create context-aware representations, while the decoder generates output sequences from these representations.

class Seq2Seq(nn.Module):

def \_\_init\_\_(self,

num\_encoder\_layers,

num\_decoder\_layers,

emb\_size,

nhead,

en\_vocab\_size,

vi\_vocab\_size,

dim\_feedforward = 512,

dropout: float = 0.1):

super(Seq2Seq, self).\_\_init\_\_()

self.transformer = nn.Transformer(d\_model=emb\_size,

nhead=nhead,

num\_encoder\_layers=num\_encoder\_layers,

num\_decoder\_layers=num\_decoder\_layers,

dim\_feedforward=dim\_feedforward,

dropout=dropout)

self.generator = nn.Linear(emb\_size, vi\_vocab\_size)

self.en\_tok\_emb = TokenEmbedding(en\_vocab\_size, emb\_size)

self.vi\_tok\_emb = TokenEmbedding(vi\_vocab\_size, emb\_size)

self.positional\_encoding = PositionalEncoding(emb\_size, dropout=dropout)

def forward(self, en, vi, en\_mask, vi\_mask, en\_padding\_mask, vi\_padding\_mask, memory\_key\_padding\_mask):

en\_emb = self.positional\_encoding(self.en\_tok\_emb(en))

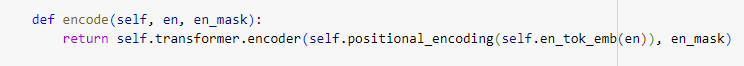
vi\_emb = self.positional\_encoding(self.vi\_tok\_emb(vi))

outs = self.transformer(en\_emb, vi\_emb, en\_mask, vi\_mask, None, en\_padding\_mask, vi\_padding\_mask, memory\_key\_padding\_mask)

return self.generator(outs)

**Encoder**: The encoder transforms the input sequence into a continuous representation that encodes contextual information. It incorporates the input token embeddings and positional encodings, followed by a series of transformer encoder layers. Each layer uses multi-head attention and feedforward networks to capture dependencies within the input sequence.

Each transformer's encoder layer includes two main components: multi head attention and feedforward network, in addition to skip connection and normalization layer. Each transformer's encoder layer includes two main components: multi head attention and feedforward network, in addition to skip connection and normalization layer.



The first sublayer implements a multi-head self-attention mechanism. We had seen that the multi-head mechanism implements heads that receive a (different) linearly projected version of the queries, keys and values each, to produce outputs in parallel that are then used to generate a final result.

**Decoder**: The decoder generates the output sequence, conditioned on the encoder's representation. It uses token embeddings and positional encodings for the target sequence, and multiple transformer decoder layers to produce the next token at each step. The decoder layers also leverage attention mechanisms to attend to the encoder's output, ensuring that generated tokens are contextually relevant.

def encode(self, en, en\_mask):

return self.transformer.encoder(self.positional\_encoding(self.en\_tok\_emb(en)), en\_mask)

def decode(self, vi, memory, vi\_mask):

return self.transformer.decoder(self.positional\_encoding(self.vi\_tok\_emb(vi)), memory, vi\_mask)

def generate\_square\_subsequent\_mask(sz):

mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0, 1)

mask = mask.float().masked\_fill(mask == 0, float('-inf')).masked\_fill(mask == 1, float(0.0))

return mask

The create\_mask function generates masks for the input and target sequences. These masks include padding masks to ignore padding tokens and subsequent masks to prevent the model from peeking into future tokens during training, ensuring autoregressive decoding.

def create\_mask(en, vi):

en\_seq\_len = en.shape[0]

vi\_seq\_len = vi.shape[0]

vi\_mask = generate\_square\_subsequent\_mask(vi\_seq\_len)

en\_mask = torch.zeros((en\_seq\_len, en\_seq\_len),device=DEVICE).type(torch.bool)

en\_padding\_mask = (en == vocabidx\_en['<pad>']).transpose(0, 1)

vi\_padding\_mask = (vi == vocabidx\_en['<pad>']).transpose(0, 1)

return en\_mask, vi\_mask, en\_padding\_mask, vi\_padding\_mask

**Greedy decode:**

def greedy\_decode(model, en, en\_mask, max\_len, start\_symbol):

en = en.to(DEVICE)

en\_mask = en\_mask.to(DEVICE)

memory = model.encode(en, en\_mask)

ys = torch.ones(1, 1).fill\_(start\_symbol).type(torch.long).to(DEVICE)

for i in range(max\_len-1):

memory = memory.to(DEVICE)

vi\_mask = (generate\_square\_subsequent\_mask(ys.size(0))

.type(torch.bool)).to(DEVICE)

out = model.decode(ys, memory, vi\_mask)

out = out.transpose(0, 1)

prob = model.generator(out[:, -1])

\_, next\_word = torch.max(prob, dim=1)

next\_word = next\_word.item()

ys = torch.cat([ys, torch.ones(1, 1).type\_as(en.data).fill\_(next\_word)], dim=0)

if next\_word == vocabidx\_en['<eos>']:

break

return ys

1. **Train model**

def train():

model = transformer.to(DEVICE)

optimizer = torch.optim.Adam(model.parameters(), lr = LR)

for epoch in range(EPOCH):

losses = 0

step = 0

for ben, bvi in train\_data:

ben = torch.tensor(ben, dtype = torch.int64).transpose(0,1).to(DEVICE)

bvi = torch.tensor(bvi, dtype = torch.int64).transpose(0,1).to(DEVICE)

vi\_input = bvi[:-1, :]

en\_mask, vi\_mask, en\_padding\_mask, vi\_padding\_mask = create\_mask(ben, vi\_input)

logits = model(ben, vi\_input, en\_mask, vi\_mask,en\_padding\_mask, vi\_padding\_mask, en\_padding\_mask)

optimizer.zero\_grad()

vi\_out = bvi[1:, :]

loss = loss\_fn(logits.reshape(-1, logits.shape[-1]), vi\_out.reshape(-1))

loss.backward()

optimizer.step()

losses = losses + loss.item()

if step % 500 == 0:

print("step:", step, "batchloss:", loss.item())

step += 1

print("epoch", epoch, "loss:", losses, '\n')

torch.save(model.state\_dict(), MODELNAME)

1. **Test model**

def test():

model.eval()

model.load\_state\_dict(torch.load('attention.model'))

ref = []

pred = []

for enprep, en, vi in test\_data:

p = translate(model, enprep)

print("INPUT: ", en)

print("REF: ", vi)

print("MT:", p, '\n')

ref.append([vi])

pred.append(p)

bleu = torchtext.data.metrics.bleu\_score(pred, ref)

print("Total:", len(test\_data))

print("BLEU:", bleu)

**Model setting**

We set this model as the following parameters

EMB\_SIZE = 512

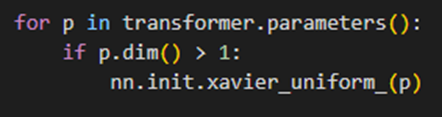
NHEAD = 8

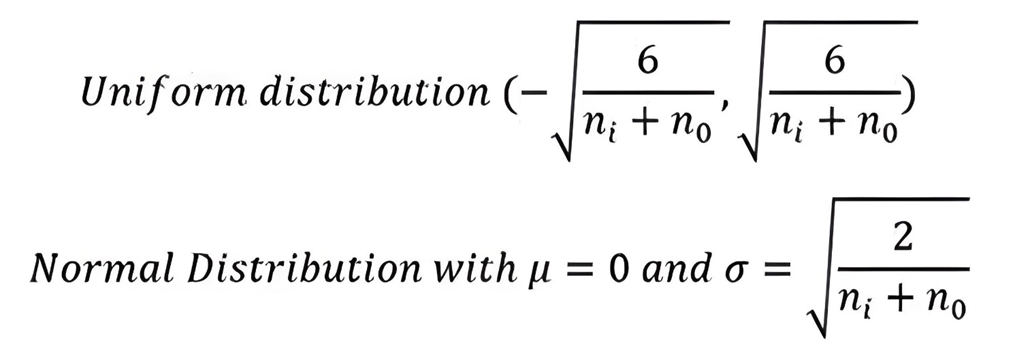
FFN\_HID\_DIM = 512

NUM\_ENCODER\_LAYERS = 2

NUM\_DECODER\_LAYERS = 2

With the passing of each layer, the Xavier initialization maintains the variance in some bounds so that we can take full advantage of the activation functions.





*Figure 2: Activation functions.*

**Execution Results:**

|  |  |
| --- | --- |
| **Before** | **After** |
| total: 1268  bleu: 0.024760886565870357 | Total: 1268  BLEU: 0.24484211843746156 |

**Explanation:**

**Seq2Seq models** addressed the issues by leveraging the power of neural networks, especially recurrent neural networks (RNN). The concept of seq2seq model was introduced in the paper titled “Sequence to Sequence Learning with Neural Networks” by Google. The architecture discussed in this research paper is a fundamental framework for natural language processing tasks. The seq2seq models are encoder-decoder models. The encoder processes the input sequence and transforms it into a fixed-size hidden representation. The decoder uses the hidden representation to generate the output sequence. The encoder-decoder structure allows them to handle input and output sequences of different lengths, making them capable of handling sequential data. Seq2Seq models are trained using a dataset of input-output pairs, where the input is a sequence of tokens, and the output is also a sequence of tokens. The model is trained to maximize the likelihood of the correct output sequence given the input sequence.

**Class PositionalEncoding:**

This class is responsible for generating positional encodings for tokens. These positional encodings help the Transformer model understand the order of tokens in a sequence. Applies dropout to mitigate overfitting.

**Class TokenEmbedding:**

This class is responsible for creating embeddings for tokens from the vocabulary.

**Class Seq2Seq:**

This class builds a Seq2Seq model using Transformer.

* **Forward pass (forward method):** Generates embeddings for input and output sentences and adds positional information using positional\_encoding. Passes these embeddings through the Transformer. Converts Transformer outputs to probabilities over the output vocabulary using a generator.
* **Encode (encode method):** Creates embeddings and adds positional information for input sentences, then passes them through the Transformer encoder.
* **Decode (decode method):** Creates embeddings and adds positional information for output sentences, then passes them through the Transformer decoder using memory from the encoder.
* **Greedy decode:** For simplicity, a Greedy Decoder is Beam search when K=1. This is necessary for inference as we don't know the: target sequence input. Therefore we try to generate the target input word by word, then feed it into the transformer.

**Supporting functions**

* **generate\_square\_subsequent\_mask:** Creates a mask to hide future positions in a sequence, preventing the model from seeing subsequent information.
* **create\_mask:** Generates masks for input and output sequences to mask padding positions and for the output sequence to mask future positions.

=> For machine translation tasks on the IWSLT15 (en-vi) dataset, **the Seq2Seq model with attention** is typically a better choice **than LSTM with dropout** due to its enhanced context capturing and ability to model complex dependencies between words in a sentence.