Table of Contents

[transformer/helpers.py 2](#_Toc195881441)

[Transformer/network.py 4](#_Toc195881442)

[transformer/train.py 10](#_Toc195881443)

[transformer/visual.py 14](#_Toc195881444)

[transformer/\_\_init\_\_,py 18](#_Toc195881445)

[train\_example\_002.py 19](#_Toc195881446)

# transformer/helpers.py

import copy

import torch

import torch.nn as nn

RUN\_EXAMPLES = True

def subsequent\_mask(size):

"Mask out subsequent positions."

attn\_shape = (1, size, size)

subsequent\_mask = torch.triu(torch.ones(attn\_shape), diagonal=1).type(

torch.uint8

)

return subsequent\_mask == 0

class LayerNorm(nn.Module):

"Construct a layernorm module (See citation for details)."

def \_\_init\_\_(self, features, eps=1e-6):

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

self.a\_2 = nn.Parameter(torch.ones(features))

self.b\_2 = nn.Parameter(torch.zeros(features))

self.eps = eps

def forward(self, x):

mean = x.mean(-1, keepdim=True)

std = x.std(-1, keepdim=True)

return self.a\_2 \* (x - mean) / (std + self.eps) + self.b\_2

def clones(module, N):

"Produce N identical layers."

return nn.ModuleList([copy.deepcopy(module) for \_ in range(N)])

def is\_interactive\_notebook():

return \_\_name\_\_ == "\_\_main\_\_"

def show\_example(fn, args=[]):

if \_\_name\_\_ == "\_\_main\_\_" and RUN\_EXAMPLES:

return fn(\*args)

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

# Transformer/network.py

import torch

import torch.nn as nn

from torch.nn.functional import log\_softmax

import math

from transformer.helpers import clones, LayerNorm

# Get ENV

from env import ENV

# Set to False to skip notebook execution (e.g. for debugging)

RUN\_EXAMPLES = True

class EncoderDecoder(nn.Module):

"""

A standard Encoder-Decoder architecture. Base for this and many

other models.

"""

def \_\_init\_\_(self, encoder, decoder, src\_embed, tgt\_embed, generator):

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\_mask, tgt, tgt\_mask)

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

return self.encoder(self.src\_embed(src), src\_mask)

def decode(self, memory, src\_mask, tgt, tgt\_mask):

return self.decoder(self.tgt\_embed(tgt), memory, src\_mask, tgt\_mask)

class Generator(nn.Module):

"Define standard linear + softmax generation step."

def \_\_init\_\_(self, d\_model, vocab):

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

self.proj = nn.Linear(d\_model, vocab)

def forward(self, x):

return log\_softmax(self.proj(x), dim=-1)

class Encoder(nn.Module):

"Core encoder is a stack of N layers"

def \_\_init\_\_(self, layer, N):

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

self.layers = clones(layer, N)

self.norm = LayerNorm(layer.size)

def forward(self, x, mask):

"Pass the input (and mask) through each layer in turn."

for layer in self.layers:

x = layer(x, mask)

return self.norm(x)

class SublayerConnection(nn.Module):

"""

A residual connection followed by a layer norm.

Note for code simplicity the norm is first as opposed to last.

"""

def \_\_init\_\_(self, size, dropout):

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

self.norm = LayerNorm(size)

self.dropout = nn.Dropout(dropout)

def forward(self, x, sublayer):

"Apply residual connection to any sublayer with the same size."

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

class EncoderLayer(nn.Module):

"Encoder is made up of self-attn and feed forward (defined below)"

def \_\_init\_\_(self, size, self\_attn, feed\_forward, dropout):

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

self.self\_attn = self\_attn

self.feed\_forward = feed\_forward

self.sublayer = clones(SublayerConnection(size, dropout), 2)

self.size = size

def forward(self, x, mask):

"Follow Figure 1 (left) for connections."

x = self.sublayer[0](x, lambda x: self.self\_attn(x, x, x, mask))

return self.sublayer[1](x, self.feed\_forward)

class Decoder(nn.Module):

"Generic N layer decoder with masking."

def \_\_init\_\_(self, layer, N):

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

self.layers = clones(layer, N)

self.norm = LayerNorm(layer.size)

def forward(self, x, memory, src\_mask, tgt\_mask):

for layer in self.layers:

x = layer(x, memory, src\_mask, tgt\_mask)

return self.norm(x)

class DecoderLayer(nn.Module):

"Decoder is made of self-attn, src-attn, and feed forward (defined below)"

def \_\_init\_\_(self, size, self\_attn, src\_attn, feed\_forward, dropout):

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

self.size = size

self.self\_attn = self\_attn

self.src\_attn = src\_attn

self.feed\_forward = feed\_forward

self.sublayer = clones(SublayerConnection(size, dropout), 3)

def forward(self, x, memory, src\_mask, tgt\_mask):

"Follow Figure 1 (right) for connections."

m = memory

x = self.sublayer[0](x, lambda x: self.self\_attn(x, x, x, tgt\_mask))

x = self.sublayer[1](x, lambda x: self.src\_attn(x, m, m, src\_mask))

return self.sublayer[2](x, self.feed\_forward)

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)) / math.sqrt(d\_k)

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

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)

nbatches = query.size(0)

# 1) Do all the linear projections in batch from d\_model => h x d\_k

query, key, value = [

lin(x).view(nbatches, -1, self.h, self.d\_k).transpose(1, 2)

for lin, x in zip(self.linears, (query, key, value))

]

# 2) Apply attention on all the projected vectors in batch.

x, self.attn = attention(

query, key, value, mask=mask, dropout=self.dropout

)

# 3) "Concat" using a view and apply a final linear.

x = (

x.transpose(1, 2)

.contiguous()

.view(nbatches, -1, self.h \* self.d\_k)

)

del query

del key

del value

return self.linears[-1](x)

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(self.w\_1(x).relu()))

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)

class PositionalEncoding(nn.Module):

"Implement the PE function."

def \_\_init\_\_(self, d\_model, dropout, max\_len=5000):

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

self.dropout = nn.Dropout(p=dropout)

# Compute the positional encodings once in log space.

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len).unsqueeze(1)

div\_term = torch.exp(

torch.arange(0, d\_model, 2) \* -(math.log(10000.0) / d\_model)

)

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0)

self.register\_buffer("pe", pe)

def forward(self, x):

x = x + self.pe[:, : x.size(1)].requires\_grad\_(False)

return self.dropout(x)

# transformer/train.py

import time

import torch

import torch.nn as nn

import pandas as pd

import altair as alt

from transformer.helpers import subsequent\_mask

class Batch:

"""Object for holding a batch of data with mask during training."""

def \_\_init\_\_(self, src, tgt=None, pad=2): # 2 = <blank>

self.src = src

self.src\_mask = (src != pad).unsqueeze(-2)

if tgt is not None:

self.tgt = tgt[:, :-1]

self.tgt\_y = tgt[:, 1:]

self.tgt\_mask = self.make\_std\_mask(self.tgt, pad)

self.ntokens = (self.tgt\_y != pad).data.sum()

@staticmethod

def make\_std\_mask(tgt, pad):

"Create a mask to hide padding and future words."

tgt\_mask = (tgt != pad).unsqueeze(-2)

tgt\_mask = tgt\_mask & subsequent\_mask(tgt.size(-1)).type\_as(

tgt\_mask.data

)

return tgt\_mask

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 steps

samples: int = 0 # total # of examples used

tokens: int = 0 # total # of tokens processed

log = lambda x, y: print(x) if y is None else y.info(x)

def run\_epoch(

data\_iter,

model,

loss\_compute,

optimizer,

scheduler,

mode="train",

accum\_iter=1,

train\_state=TrainState(),

logger=None

):

"""Train a single epoch"""

start = time.time()

total\_tokens = 0

total\_loss = 0

tokens = 0

n\_accum = 0

for i, batch in enumerate(data\_iter):

out = model.forward(

batch.src, batch.tgt, batch.src\_mask, batch.tgt\_mask

)

loss, loss\_node = loss\_compute(out, batch.tgt\_y, batch.ntokens)

# loss\_node = loss\_node / accum\_iter

if mode == "train" or mode == "train+log":

loss\_node.backward()

train\_state.step += 1

train\_state.samples += batch.src.shape[0]

train\_state.tokens += batch.ntokens

if i % accum\_iter == 0:

optimizer.step()

optimizer.zero\_grad(set\_to\_none=True)

n\_accum += 1

train\_state.accum\_step += 1

scheduler.step()

total\_loss += loss

total\_tokens += batch.ntokens

tokens += batch.ntokens

if i % 40 == 1 and (mode == "train" or mode == "train+log"):

lr = optimizer.param\_groups[0]["lr"]

elapsed = time.time() - start

log(

(

"Epoch Step: %6d | Accumulation Step: %3d | Loss: %6.2f "

+ "| Tokens / Sec: %7.1f | Learning Rate: %6.1e"

)

% (i, n\_accum, loss / batch.ntokens, tokens / elapsed, lr), logger

)

start = time.time()

tokens = 0

del loss

del loss\_node

return total\_loss / total\_tokens, train\_state

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 \* warmup \*\* (-1.5))

)

class LabelSmoothing(nn.Module):

"Implement label smoothing."

def \_\_init\_\_(self, size, padding\_idx, smoothing=0.0):

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

self.criterion = nn.KLDivLoss(reduction="sum")

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, true\_dist.clone().detach())

def loss(x, crit):

d = x + 3 \* 1

predict = torch.FloatTensor([[0, x / d, 1 / d, 1 / d, 1 / d]])

return crit(predict.log(), torch.LongTensor([1])).data

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.contiguous().view(-1)

)

/ norm

)

return sloss.data \* norm, sloss

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

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

ys = torch.zeros(1, 1).fill\_(start\_symbol).type\_as(src.data)

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

out = model.decode(

memory, src\_mask, ys, subsequent\_mask(ys.size(1)).type\_as(src.data)

)

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

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

next\_word = next\_word.data[0]

ys = torch.cat(

[ys, torch.zeros(1, 1).type\_as(src.data).fill\_(next\_word)], dim=1

)

return ys

# transformer/visual.py

import pandas as pd

import altair as alt

import torch

def mtx2df(m, max\_row, max\_col, row\_tokens, col\_tokens):

"convert a dense matrix to a data frame with row and column indices"

return pd.DataFrame(

[

(

r,

c,

float(m[r, c]),

"%.3d %s"

% (r, row\_tokens[r] if len(row\_tokens) > r else "<blank>"),

"%.3d %s"

% (c, col\_tokens[c] if len(col\_tokens) > c else "<blank>"),

)

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\_col],

columns=["row", "column", "value", "row\_token", "col\_token"],

)

def attn\_map(attn, layer, head, row\_tokens, col\_tokens, max\_dim=30):

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",

tooltip=["row", "column", "value", "row\_token", "col\_token"],

)

.properties(height=400, width=400)

.interactive()

)

def get\_encoder(model, layer):

return model.encoder.layers[layer].self\_attn.attn

def get\_decoder\_self(model, layer):

return model.decoder.layers[layer].self\_attn.attn

def get\_decoder\_src(model, layer):

return model.decoder.layers[layer].src\_attn.attn

def visualize\_layer(model, layer, getter\_fn, ntokens, row\_tokens, col\_tokens):

# 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))

def viz\_encoder\_self(model, example\_data):

example = example\_data[

len(example\_data) - 1

] # batch object for the final example

layer\_viz = [

visualize\_layer(

model, layer, get\_encoder, 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]

)

def viz\_decoder\_self(model, example\_data):

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]

)

def viz\_decoder\_src(model, example\_data):

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]

)

# transformer/\_\_init\_\_,py

import torch

import torch.nn as nn

from torchinfo import summary

import copy

from transformer.network import MultiHeadedAttention, PositionwiseFeedForward, PositionalEncoding, EncoderDecoder, Encoder, EncoderLayer, Decoder, DecoderLayer, Embeddings, Generator

log = lambda x, y: print(x) if y is None else y.info(x)

def make\_model(

src\_vocab,

tgt\_vocab,

N=6,

d\_model=512,

d\_ff=2048,

h=8,

dropout=0.1,

logger=None

):

"Helper: Construct a model from hyperparameters."

c = copy.deepcopy

attn = MultiHeadedAttention(h, d\_model)

ff = PositionwiseFeedForward(d\_model, d\_ff, dropout)

position = PositionalEncoding(d\_model, dropout)

model = EncoderDecoder(

Encoder(EncoderLayer(d\_model, c(attn), c(ff), dropout), N),

Decoder(DecoderLayer(d\_model, c(attn), c(attn), c(ff), dropout), N),

nn.Sequential(Embeddings(d\_model, src\_vocab), c(position)),

nn.Sequential(Embeddings(d\_model, tgt\_vocab), c(position)),

Generator(d\_model, tgt\_vocab),

)

log(summary(model, verbose=0), logger)

# This was important from their code.

# Initialize parameters with Glorot / fan\_avg.

for p in model.parameters():

if p.dim() > 1:

nn.init.xavier\_uniform\_(p)

return model

# train\_example\_002.py

from os.path import exists

import torch

from torch.optim.lr\_scheduler import LambdaLR

import spacy

import torchtext.datasets as datasets

from torchtext.vocab import build\_vocab\_from\_iterator

from torch.nn.parallel import DistributedDataParallel as DDP

import torch.distributed as dist

from torch.utils.data import DataLoader

from torch.nn.functional import pad

from torchtext.data.functional import to\_map\_style\_dataset

from torch.utils.data.distributed import DistributedSampler

from torchinfo import summary

import GPUtil

import torch.multiprocessing as mp

import altair as alt

# Get ENV

from env import ENV, ENV\_LOCAL\_RC

from transformer import make\_model

from transformer.train import Batch, LabelSmoothing, rate, run\_epoch, TrainState, SimpleLossCompute, greedy\_decode

from transformer.helpers import DummyOptimizer, DummyScheduler

import transformer.visual as visual

from common.logger import FileLogger

from prepare import generate\_new\_resultdir, copy\_env, dump\_hyper\_params

RESULT\_DIR = generate\_new\_resultdir(framework="transformer")

VOCAB\_PT\_FILEPATH = os.path.join(RESULT\_DIR, "vocab.pt")

MODEL\_PT\_FILEPATH = os.path.join(RESULT\_DIR, "multi30k\_model\_final.pt")

logger = FileLogger(os.path.join(RESULT\_DIR, "train.log"))

logger.info("RESULT\_DIR %s" % RESULT\_DIR)

# To display altair with browser

# TODO, further save these images to disk RESULT dir

alt.renderers.enable("browser")

# Load spacy tokenizer models, download them if they haven't been

# 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])

def collate\_batch(

batch,

src\_pipeline,

tgt\_pipeline,

src\_vocab,

tgt\_vocab,

device,

max\_padding=128,

pad\_id=2,

):

bs\_id = torch.tensor([0], device=device) # <s> token id

eos\_id = torch.tensor([1], device=device) # </s> token id

src\_list, tgt\_list = [], []

for (\_src, \_tgt) in batch:

processed\_src = torch.cat(

[

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 of padding - len

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,

)

)

src = torch.stack(src\_list)

tgt = torch.stack(tgt\_list)

return (src, tgt)

def create\_dataloaders(

device,

vocab\_src,

vocab\_tgt,

spacy\_de,

spacy\_en,

batch\_size=12000,

max\_padding=128,

is\_distributed=True,

):

# def create\_dataloaders(batch\_size=12000):

def tokenize\_de(text):

return tokenize(text, spacy\_de)

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>"],

)

# downloaded to C:\Users\Administrator\.cache\torch\text\datasets\Multi30k automactically.

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 else None

)

valid\_iter\_map = to\_map\_style\_dataset(valid\_iter)

valid\_sampler = (

DistributedSampler(valid\_iter\_map) if is\_distributed else None

)

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),

sampler=valid\_sampler,

collate\_fn=collate\_fn,

)

return train\_dataloader, valid\_dataloader

def build\_vocabulary(spacy\_de, spacy\_en):

def tokenize\_de(text):

return tokenize(text, spacy\_de)

def tokenize\_en(text):

return tokenize(text, spacy\_en)

logger.info("Building German Vocabulary ...")

train, val, test = datasets.Multi30k(language\_pair=("de", "en"))

vocab\_src = build\_vocab\_from\_iterator(

yield\_tokens(train + val + test, tokenize\_de, index=0),

min\_freq=2,

specials=["<s>", "</s>", "<blank>", "<unk>"],

)

logger.info("Building English Vocabulary ...")

train, val, test = datasets.Multi30k(language\_pair=("de", "en"))

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\_FILEPATH):

vocab\_src, vocab\_tgt = build\_vocabulary(spacy\_de, spacy\_en)

torch.save((vocab\_src, vocab\_tgt), VOCAB\_PT\_FILEPATH)

else:

vocab\_src, vocab\_tgt = torch.load(VOCAB\_PT\_FILEPATH)

logger.info("Finished.\nVocabulary sizes:")

logger.info(len(vocab\_src))

logger.info(len(vocab\_tgt))

return vocab\_src, vocab\_tgt

def train\_worker(

gpu,

ngpus\_per\_node,

vocab\_src,

vocab\_tgt,

spacy\_de,

spacy\_en,

hyper\_params,

is\_distributed=False,

):

logger.info(f"Train worker process using GPU: {gpu} for training")

torch.cuda.set\_device(gpu)

pad\_idx = vocab\_tgt["<blank>"]

d\_model = 512

model = make\_model(len(vocab\_src), len(vocab\_tgt), N=6,logger=logger)

model.cuda(gpu)

module = model

is\_main\_process = True

if is\_distributed:

dist.init\_process\_group(

"nccl", init\_method="env://", rank=gpu, world\_size=ngpus\_per\_node

)

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=0.1

)

criterion.cuda(gpu)

train\_dataloader, valid\_dataloader = create\_dataloaders(

gpu,

vocab\_src,

vocab\_tgt,

spacy\_de,

spacy\_en,

batch\_size=hyper\_params["batch\_size"] // ngpus\_per\_node,

max\_padding=hyper\_params["max\_padding"],

is\_distributed=is\_distributed,

)

optimizer = torch.optim.Adam(

model.parameters(), lr=hyper\_params["base\_lr"], betas=(0.9, 0.98), eps=1e-9

)

lr\_scheduler = LambdaLR(

optimizer=optimizer,

lr\_lambda=lambda step: rate(

step, d\_model, factor=1, warmup=hyper\_params["warmup"]

),

)

train\_state = TrainState()

for epoch in range(hyper\_params["num\_epochs"]):

if is\_distributed:

train\_dataloader.sampler.set\_epoch(epoch)

valid\_dataloader.sampler.set\_epoch(epoch)

model.train()

logger.info(f"[GPU{gpu}] Epoch {epoch} Training ====")

\_, train\_state = run\_epoch(

(Batch(b[0], b[1], pad\_idx) for b in train\_dataloader),

model,

SimpleLossCompute(module.generator, criterion),

optimizer,

lr\_scheduler,

mode="train+log",

accum\_iter=hyper\_params["accum\_iter"],

train\_state=train\_state,

logger=logger

)

GPUtil.showUtilization()

# skip temp model files to save space and boost time

# if is\_main\_process:

# file\_path = os.path.join(RESULT\_DIR, "%s%.2d.pt" % (hyper\_params["file\_prefix"], epoch))

# torch.save(module.state\_dict(), file\_path)

torch.cuda.empty\_cache()

logger.info(f"[GPU{gpu}] Epoch {epoch} Validation ====")

model.eval()

sloss = run\_epoch(

(Batch(b[0], b[1], pad\_idx) for b in valid\_dataloader),

model,

SimpleLossCompute(module.generator, criterion),

DummyOptimizer(),

DummyScheduler(),

mode="eval",

)

logger.info(sloss)

torch.cuda.empty\_cache()

if is\_main\_process:

torch.save(module.state\_dict(), MODEL\_PT\_FILEPATH)

def train\_distributed\_model(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, hyper\_params):

ngpus = torch.cuda.device\_count()

os.environ["MASTER\_ADDR"] = "localhost"

os.environ["MASTER\_PORT"] = "12356"

logger.info(f"Number of GPUs detected: {ngpus}")

logger.info("Spawning training processes ...")

mp.spawn(

train\_worker,

nprocs=ngpus,

args=(ngpus, vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, hyper\_params, True),

)

def train\_model(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, hyper\_params):

if hyper\_params["distributed"]:

train\_distributed\_model(

vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, hyper\_params

)

else:

train\_worker(

0, 1, vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, hyper\_params, False

)

def load\_trained\_model(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en):

hyper\_params = {

"batch\_size": ENV.int("HYPER\_PARAM\_BATCHSIZE", 32),

"distributed": ENV.bool("HYPER\_PARAM\_DISTRIBUTED", False),

"num\_epochs": ENV.int("HYPER\_PARAM\_EPOCH", 10),

"accum\_iter": 10,

"base\_lr": 1.0,

"max\_padding": 72,

"warmup": 3000,

"file\_prefix": "multi30k\_model\_",

}

dump\_hyper\_params(hyper\_params,resultdir=RESULT\_DIR)

if not exists(MODEL\_PT\_FILEPATH):

copy\_env(ENV\_LOCAL\_RC, RESULT\_DIR)

train\_model(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, hyper\_params)

model = make\_model(len(vocab\_src), len(vocab\_tgt), N=6)

model.load\_state\_dict(torch.load(MODEL\_PT\_FILEPATH))

return model

def check\_outputs(

valid\_dataloader,

model,

vocab\_src,

vocab\_tgt,

n\_examples=15,

pad\_idx=2,

eos\_string="</s>",

):

results = [()] \* n\_examples

for idx in range(n\_examples):

logger.info("\nExample %d ========\n" % idx)

b = next(iter(valid\_dataloader))

rb = Batch(b[0], b[1], pad\_idx)

greedy\_decode(model, rb.src, rb.src\_mask, 64, 0)[0]

src\_tokens = [

vocab\_src.get\_itos()[x] for x in rb.src[0] if x != pad\_idx

]

tgt\_tokens = [

vocab\_tgt.get\_itos()[x] for x in rb.tgt[0] if x != pad\_idx

]

logger.info(

"Source Text (Input) : "

+ " ".join(src\_tokens).replace("\n", "")

)

logger.info(

"Target Text (Ground Truth) : "

+ " ".join(tgt\_tokens).replace("\n", "")

)

model\_out = greedy\_decode(model, rb.src, rb.src\_mask, 72, 0)[0]

model\_txt = (

" ".join(

[vocab\_tgt.get\_itos()[x] for x in model\_out if x != pad\_idx]

).split(eos\_string, 1)[0]

+ eos\_string

)

logger.info("Model Output : " + model\_txt.replace("\n", ""))

results[idx] = (rb, src\_tokens, tgt\_tokens, model\_out, model\_txt)

return results

def run\_model\_example(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, device="cpu", n\_examples=5):

logger.info("Preparing Data ...")

\_, valid\_dataloader = create\_dataloaders(

torch.device("cpu"),

vocab\_src,

vocab\_tgt,

spacy\_de,

spacy\_en,

batch\_size=1,

is\_distributed=False,

)

logger.info("Loading Trained Model ...")

model = make\_model(len(vocab\_src), len(vocab\_tgt), N=6)

model.load\_state\_dict(

torch.load(MODEL\_PT\_FILEPATH, map\_location=torch.device(device))

)

logger.info("Checking Model Outputs:")

example\_data = check\_outputs(

valid\_dataloader, model, vocab\_src, vocab\_tgt, n\_examples=n\_examples

)

return model, example\_data

def main():

spacy\_de, spacy\_en = load\_tokenizers()

vocab\_src, vocab\_tgt = load\_vocab(spacy\_de, spacy\_en)

model = load\_trained\_model(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en)

'''

# Additional Components: BPE, Search, Averaging

2. Shared Embeddings: When using BPE with shared vocabulary we can share the same weight vectors between the source / target / generator. See the (cite) for details. To add this to the model simply do this:

'''

if False:

model.src\_embed[0].lut.weight = model.tgt\_embeddings[0].lut.weight

model.generator.lut.weight = model.tgt\_embed[0].lut.weight

'''

> 4) Model Averaging: The paper averages the last k checkpoints to

> create an ensembling effect. We can do this after the fact if we

> have a bunch of models:

'''

def average(model, models):

"Average models into model"

for ps in zip(\*[m.params() for m in [model] + models]):

ps[0].copy\_(torch.sum(\*ps[1:]) / len(ps[1:]))

logger.info("Preparing Data ...")

\_, valid\_dataloader = create\_dataloaders(

torch.device("cuda"),

vocab\_src,

vocab\_tgt,

spacy\_de,

spacy\_en,

batch\_size=1,

is\_distributed=False,

)

model, example\_data = run\_model\_example(vocab\_src, vocab\_tgt, spacy\_de, spacy\_en, device="cpu", n\_examples=5)

visual.viz\_encoder\_self(model, example\_data).show()

visual.viz\_decoder\_self(model, example\_data).show()

visual.viz\_decoder\_src(model, example\_data).show()

if \_\_name\_\_ == '\_\_main\_\_':

main()