Transformer

① Define the model

✲ Language modeling task

: to assign probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words.

- Sequence of tokens $\xrightarrow[\text{passed to}]{}$ embedding layer

↓

- positional encoding layer

↓

- Transformer Encoder
  - self-attention layer
    (square attention mask required)

↓

- Linear layer

- log-Softmax function

② Load & batch data

- batchify() function

  : arranges the dataset into columns,
  trimming off any tokens remaining
  after the data has been divided into batches of size batch_size

e.g.

[A, B, C, D, ... X, Y, Z] $\xrightarrow[\text{4}]{\text{batch size}}$

Alphabet as the sequence
(total length of 26)

A	G	·	
B	H	·	
C	I		:
D	J		:
E	K		W
F	L		X

4 sequences of length 6

```python
import math

import torch
import torch.nn as nn
import torch.nn.functional as F

from torch.nn import TransformerEncoder, TransformerEncoderLayer

class TransformerModel(nn.Module):

    def __init__(self, ntoken: int, d_model: int, nhead: int, d_hid: int,
                 nlayers: int, dropout: float = 0.5):
        super().__init__()
        self.model_type = 'Transformer'
        self.pos_encoding = PositionalEncoding(d_model, dropout)
        encoder_layers = TransformerEncoderLayer(d_model, nhead, d_hid, dropout)
        self.encoder = nn.Embedding(ntoken, d_model)
        self.d_model = d_model
        self.decoder = nn.Linear(d_model, ntoken)

        self.init_weights()

    def init_weights(self) -> None:
        initrange = 0.1
        self.encoder.weight.data.uniform_(-initrange, initrange)
        self.decoder.bias.data.zero_()
        self.decoder.weight.data.uniform_(-initrange, initrange)

    def generate_square_subsequent_mask(sz: int) -> Tensor:
        return torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal = 1)

    def forward(self, src, src_mask):
        src = self.encoder(src) * math.sqrt(self.ninp)
        src = self.pos_encoder(src)
        output = self.transformer_encoder(src, src_mask)
        output = self.decoder(output)

        return output
```

```
Class PositionalEncoding (nn.Module):

    def __init__(self, d_model, dropout = 0.1, max_len = 5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p = dropout)

        pe = torch.zeros(max_len, 1, d_model)
        position = torch.arange(0, max_len, dtype = torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                             (- math.log(10000.0) / d_model))
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.*register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

Load & batch data

```
import torch
from torchtext.datasets import WikiText2
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator

train_iter = WikiText2(split='train')
tokenizer = get_tokenizer('basic_english')
vocab = build_vocab_from_iterator(map(tokenizer, train_iter), specials=["<unk>"])
vocab.set_default_index(vocab["<unk>"])

def data_process(raw_text_iter):
    data = [torch.tensor(vocab(tokenizer(item)), dtype=torch.long) for item in raw_text_iter]
    return torch.cat(tuple(filter(lambda t: t.numel() > 0, data)))

train_iter, val_iter, test_iter = WikiText2()
train_data = data_process(train_iter)
val_data = data_process(val_iter)
test_data = data_process(test_iter)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def batchify(data, bsz):
    # Divide the dataset into bsz parts.
    seq_len = data.size(0) // bsz

    # Trim off any extra elements that wouldn't cleanly fit (remainders)
    data = data[:seq_len * bsz]

    # Evenly divide the data across the bsz batches.
    data = data.view(bsz, seq_len).t().contiguous()
    return data.to(device)

batch_size = 20
eval_batch_size = 10
train_data = batchify(train_data, batch_size)
val_data = batchify(val_data, eval_batch_size)
test_data = batchify(test_data, eval_batch_size)
```

get_batch()

- generates a pair of input-target sequences for the transformer model.
- it subdivides the source data into chunks of length bptt.

```
bptt = 35

def get_batch (source, i):
    seq_len = min (bptt, len (source) -1 - i)
    data = source [i : i+ seq_len]
    target = source [i+1 : i+1 + seq_len]. reshape (-1)
    return data, target
```

Initiate an instance

```
ntokens  =  len(vocab)      # the size of vocabulary
emsize  =  200              # embedding dimension
d_hid =  200                # the dimension of feedforward network model in nn.TransformerEncoder
nlayers  = 2                # the number of nn.TransformerEncoderLayer in nn.TransformerEncoder
nhead = 2                   # the number of heads in the multihead attention models
dropout  = 0.2              # the dropout value
model = TransformerModel (ntokens, emsize, nhead, n_hid, nlayers, dropout).to(device)
```

Run the model

```
import time

criterion = nn.CrossEntropyLoss()
lr = 5.0
optimizer = torch.optim.SGD(model.parameters(), lr = lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma = 0.95)

def train(model: nn.Module) → None:
    model.train()
    total_loss = 0.
    log_interval = 200
    start_time = time.time()
    src_mask = model.generate_square_subsequent_mask(bptt).to(device)
    num_batches = len(train_data) // bptt

    for batch, i in enumerate(range(0, train_data.size(0) - 1, bptt)):
        data, targets = get_batch(train_data, i)
        batch_size = data.size(0)
        if batch_size != bptt:
            src_mask = src_mask[:batch_size, :batch_size]
        output = model(data, src_mask)
        loss = criterion(output.view(-1, ntokens), targets)

        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.*clip_grad_norm_(model.parameters(), 0.5)
        optimizer.step()

        total_loss += loss.item()
        if batch % log_interval == 0 and batch > 0:
            lr = scheduler.get_last_lr()[0]
            ms_per_batch = (time.time() - start.time) * 1000 / log_interval
            cur_loss = total_loss / log_interval
            ppl = math.exp(cur_loss)
            print(f'| epoch {epoch:3d} | {batch:5d} / {num_batches:5d} batches |'
                  f'lr {lr:02.2f} | ms/batch {ms_per_batch:5.2f} |'
                  f'loss {cur_loss:*5.2f} | ppl {ppl:8.2f}')
            total_loss = 0
            start_time = time.time()
```

```python
def evaluate(model: nn.Module, eval_data: Tensor) -> float:
    model.eval()    # turn on evaluation mode.
    total_loss = 0.
    src_mask = generate_square_subsequent_mask(bptt).to(device)
    with torch.no_grad():
        for i in range(0, eval_data.size(0) - 1, bptt):
            data, targets = get_batch(eval_data, i)
            batch_size = data.size(0)
            if batch_size != bptt:
                src_mask = src_mask[:batch_size, :batch_size]
            output = model(data, src_mask)
            output_flat = output.view(-1, ntokens)
            total_loss += batch_size * criterion(output_flat, targets).item()
    return total_loss / (len(eval_data) - 1)
```

Loop over epochs

```python
best_val_loss = float('inf')
epoch = 3
best_model = None

for epoch in range(1, epochs + 1):
    epoch_start_time = time.time()
    train(model)
    val_loss = evaluate(model, val_data)
    val_ppl = math.exp(val_loss)
    elapsed = time.time() - epoch_start_time
    print('-' * 89)
    print(f'| end of epoch {epoch:3d} | time: {elapsed:5.2f}s |'
          f'valid loss {val_loss:5.2f} | valid ppl {val_ppl:8.2f}')
    print('-' * 89)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model = copy.deepcopy(model)

    scheduler.step()
```

Evaluate the best model on the test dataset

```
test_loss = evaluate(best_model, test_data)
test_ppl = math.exp(test_loss)
print('=' * 89)
print(f'| End of training | test loss {test_loss:5.2f} | '
      f'test ppl {test_ppl:8.2f}')
print('=' * 89)
```