Main.py

```python
# coding: utf-8
import argparse
import time
import math
import os
import torch
import torch.nn as nn

import data
import model

parser = argparse.ArgumentParser(description='PyTorch Wikitext-2 RNN/LSTM/GRU/Transformer Lan
parser.add_argument('--data', type=str, default='./data/wikitext-2',
                    help='location of the data corpus')
parser.add_argument('--model', type=str, default='LSTM',
                    help='type of network (RNN_TANH, RNN_RELU, LSTM, GRU, Transformer)')
parser.add_argument('--emsize', type=int, default=200,
                    help='size of word embeddings')
parser.add_argument('--nhid', type=int, default=200,
                    help='number of hidden units per layer')
parser.add_argument('--nlayers', type=int, default=2,
                    help='number of layers')
parser.add_argument('--lr', type=float, default=20,
                    help='initial learning rate')
parser.add_argument('--clip', type=float, default=0.25,
                    help='gradient clipping')
parser.add_argument('--epochs', type=int, default=40,
                    help='upper epoch limit')
parser.add_argument('--batch_size', type=int, default=20, metavar='N',
                    help='batch size')
```

```python
parser.add_argument('--bptt', type=int, default=35,
                    help='sequence length')
parser.add_argument('--dropout', type=float, default=0.2,
                    help='dropout applied to layers (0 = no dropout)')
parser.add_argument('--tied', action='store_true',
                    help='tie the word embedding and softmax weights')
parser.add_argument('--seed', type=int, default=1111,
                    help='random seed')
parser.add_argument('--cuda', action='store_true',
                    help='use CUDA')
parser.add_argument('--log-interval', type=int, default=200, metavar='N',
                    help='report interval')
parser.add_argument('--save', type=str, default='model.pt',
                    help='path to save the final model')
parser.add_argument('--nhead', type=int, default=2,
                    help='the number of heads in the encoder/decoder of the transformer model
parser.add_argument('--dry-run', action='store_true',
                    help='verify the code and the model')
args = parser.parse_args()

# Set the random seed manually for reproducibility.
torch.manual_seed(args.seed)
if torch.cuda.is_available():
    if not args.cuda:
        print("WARNING: You have a CUDA device, so you should probably run with --cuda.")

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

```

```
59  ###############################################################################
60  # Load data
61  ###############################################################################
62
63  corpus = data.Corpus(args.data)
64  # Starting from sequential data, batchify arranges the dataset into columns.
65  # For instance, with the alphabet as the sequence and batch size 4, we'd get
66  # ┌ a g m s ┐
67  # | b h n t |
68  # | c i o u |
69  # | d j p v |
70  # | e k q w |
71  # └ f l r x ┘.
72  # These columns are treated as independent by the model, which means that the
73  # dependence of e. g. 'g' on 'f' can not be learned, but allows more efficient
74  # batch processing.
75
76  def batchify(data, bsz):
77      # Work out how cleanly we can divide the dataset into bsz parts.
78      nbatch = data.size(0) // bsz
79      # Trim off any extra elements that wouldn't cleanly fit (remainders).
80      data = data.narrow(0, 0, nbatch * bsz)
81      # Evenly divide the data across the bsz batches.
82      data = data.view(bsz, -1).t().contiguous()
83      return data.to(device)
84
85  eval_batch_size = 10
86  train_data = batchify(corpus.train, args.batch_size)
87  val_data = batchify(corpus.valid, eval_batch_size)
88  test_data = batchify(corpus.test, eval_batch_size)
```

```
90  ###############################################################################
91  # Build the model
92  ###############################################################################
93
94  ntokens = len(corpus.dictionary)
95  if args.model == 'LSTM':
96      model = model.RNNModel(args.model, ntokens, args.emsize, args.nhid, args.nlayers, args.dr
97
98  criterion = nn.CrossEntropyLoss()
99
100 ###############################################################################
101 # Training code
102 ###############################################################################
103
104 def repackage_hidden(h):
105     """Wraps hidden states in new Tensors, to detach them from their history."""
106
107     if isinstance(h, torch.Tensor):
108         return h.detach()
109     else:
110         return tuple(repackage_hidden(v) for v in h)
111
112
113 # get_batch subdivides the source data into chunks of length args.bptt.
114 # If source is equal to the example output of the batchify function, with
115 # a bptt-limit of 2, we'd get the following two Variables for i = 0:
116 # ┌ a g m s ┐ ┌ b h n t ┐
117 # └ b h n t ┘ └ c i o u ┘
118 # Note that despite the name of the function, the subdivison of data is not
119 # done along the batch dimension (i.e. dimension 1), since that was handled
```

```
# by the batchify function. The chunks are along dimension 0, corresponding
# to the seq_len dimension in the LSTM.

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


def evaluate(data_source):
    # Turn on evaluation mode which disables dropout.
    model.eval()
    total_loss = 0.
    ntokens = len(corpus.dictionary)
    hidden = model.init_hidden(eval_batch_size)
    with torch.no_grad():
        for i in range(0, data_source.size(0) - 1, args.bptt):
            data, targets = get_batch(data_source, i)
            output, hidden = model(data, hidden)
            hidden = repackage_hidden(hidden)
            total_loss += len(data) * criterion(output, targets).item()
    return total_loss / (len(data_source) - 1)
```

```
def train():
    # Turn on training mode which enables dropout.
    model.train()
    total_loss = 0.
    start_time = time.time()
    ntokens = len(corpus.dictionary)
    hidden = model.init_hidden(args.batch_size)
    for batch, i in enumerate(range(0, train_data.size(0) - 1, args.bptt)):
        data, targets = get_batch(train_data, i)
        # Starting each batch, we detach the hidden state from how it was previously produced
        # If we didn't, the model would try backpropagating all the way to start of the datas
        model.zero_grad()
        hidden = repackage_hidden(hidden)
        output, hidden = model(data, hidden) # model.forward(data, hidden)
        loss = criterion(output, targets)
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        torch.nn.utils.clip_grad_norm_(model.parameters(), args.clip)

        for p in model.parameters():
            p.data.add_(p.grad, alpha=-lr)
            # optimizer: admw adafactor ...
        total_loss += loss.item()
```

```
        if batch % args.log_interval == 0 and batch > 0:
            cur_loss = total_loss / args.log_interval
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches | lr {:02.2f} | ms/batch {:5.2f} | '
                    'loss {:5.2f} | ppl {:8.2f}'.format(
                epoch, batch, len(train_data) // args.bptt, lr,
                elapsed * 1000 / args.log_interval, cur_loss, math.exp(cur_loss)))
            total_loss = 0
            start_time = time.time()
        if args.dry_run:
            break

# Loop over epochs.
lr = args.lr
best_val_loss = None
```

```
184
185 # At any point you can hit Ctrl + C to break out of training early.
186
187 try:
188     for epoch in range(1, args.epochs+1):
189         epoch_start_time = time.time()
190         train()
191         val_loss = evaluate(val_data)
192         print('-' * 89)
193         print('| end of epoch {:3d} | time: {:5.2f}s | valid loss {:5.2f} | '
194                 'valid ppl {:8.2f}'.format(epoch, (time.time() - epoch_start_time),
195                                            val_loss, math.exp(val_loss)))
196         print('-' * 89)
197         # Save the model if the validation loss is the best we've seen so far.
198         if not best_val_loss or val_loss < best_val_loss:
199             with open(args.save, 'wb') as f:
200                 torch.save(model, f)
201             best_val_loss = val_loss
202         else:
203             # Anneal the learning rate if no improvement has been seen in the validation data
204             lr /= 4.0
205 except KeyboardInterrupt:
206     print('-' * 89)
207     print('Exiting from training early')
```

```
210 # Load the best saved model.
211 with open(args.save, 'rb') as f:
212     model = torch.load(f)
213     # after load the rnn params are not a continuous chunk of memory
214     # this makes them a continuous chunk, and will speed up forward pass
215     # Currently, only rnn model supports flatten_parameters function.
216     if args.model in ['RNN_TANH', 'RNN_RELU', 'LSTM', 'GRU']:
217         model.rnn.flatten_parameters()
218
219 # Run on test data.
220 test_loss = evaluate(test_data)
221 print('=' * 89)
222 print('| End of training | test loss {:5.2f} | test ppl {:8.2f}'.format(
223     test_loss, math.exp(test_loss)))
224 print('=' * 89)
```

Data.py

```
1  import os
2  from io import open
3  import torch
4
5  class Dictionary(object):
6      def __init__(self):
7          self.word2idx = {}
8          self.idx2word = []
9
10     def add_word(self, word):
11         if word not in self.word2idx:
12             self.idx2word.append(word)
13             self.word2idx[word] = len(self.idx2word) - 1
14         return self.word2idx[word]
15
16     def __len__(self):
17         return len(self.idx2word)
```

```python
class Corpus(object):
    def __init__(self, path):
        self.dictionary = Dictionary()
        self.train = self.tokenize(os.path.join(path, 'train.txt'))
        self.valid = self.tokenize(os.path.join(path, 'valid.txt'))
        self.test = self.tokenize(os.path.join(path, 'test.txt'))

    def tokenize(self, path):
        """Tokenizes a text file."""
        assert os.path.exists(path)
        # Add words to the dictionary
        with open(path, 'r', encoding="utf8") as f:
            for line in f:
                words = line.split() + ['<eos>']
                for word in words:
                    self.dictionary.add_word(word)

        # Tokenize file content
        with open(path, 'r', encoding="utf8") as f:
            idss = []
            for line in f:
                words = line.split() + ['<eos>']
                ids = []
                for word in words:
                    ids.append(self.dictionary.word2idx[word])
                idss.append(torch.tensor(ids).type(torch.int64))
            ids = torch.cat(idss)

        return ids
```

Model.py

```python
import math
import torch
import torch.nn as nn
import torch.nn.functional as F

class RNNModel(nn.Module):
    """Container module with an encoder, a recurrent module, and a decoder."""

    def __init__(self, rnn_type, ntoken, ninp, nhid, nlayers, dropout=0.5, tie_weights=False)
        super(RNNModel, self).__init__()
        self.ntoken = ntoken
        self.encoder = nn.Embedding(ntoken, ninp)
        if rnn_type in ['LSTM', 'GRU']:
            self.rnn = getattr(nn, rnn_type)(ninp, nhid, nlayers, dropout=dropout)
        else:
            try:
                nonlinearity = {'RNN_TANH': 'tanh', 'RNN_RELU': 'relu'}[rnn_type]
            except KeyError:
                raise ValueError( """An invalid option for `--model` was supplied,
                                     options are ['LSTM', 'GRU', 'RNN_TANH' or 'RNN_RELU']""")
            self.rnn = nn.RNN(ninp, nhid, nlayers, nonlinearity=nonlinearity, dropout=dropout
        self.decoder = nn.Linear(nhid, ntoken)

        self.drop = nn.Dropout(dropout)

        self.init_weights()

        self.rnn_type = rnn_type
        self.nhid = nhid
        self.nlayers = nlayers
```

```python
    def init_weights(self):
        initrange = 0.1
        nn.init.uniform_(self.encoder.weight, -initrange, initrange)
        nn.init.zeros_(self.decoder.bias)
        nn.init.uniform_(self.decoder.weight, -initrange, initrange)

    def forward(self, input, hidden):
        emb = self.drop(self.encoder(input))
        output, hidden = self.rnn(emb, hidden)
        output = self.drop(output)
        decoded = self.decoder(output)
        decoded = decoded.view(-1, self.ntoken)
        return decoded, hidden

    def init_hidden(self, bsz):
        weight = next(self.parameters())
        if self.rnn_type == 'LSTM':
            return (weight.new_zeros(self.nlayers, bsz, self.nhid),
                    weight.new_zeros(self.nlayers, bsz, self.nhid))
        else:
            return weight.new_zeros(self.nlayers, bsz, self.nhid)
```

.

参考训练结果

```
| epoch   1 |  1000/ 2983 batches | lr 20.00 | ms/batch 33.42 | loss  6.68 | ppl   797.21
| epoch   1 |  2000/ 2983 batches | lr 20.00 | ms/batch 13.90 | loss  5.90 | ppl   366.17
-----------------------------------------------------------------------------------------
| end of epoch   1 | time: 62.67s | valid loss  5.52 | valid ppl   249.07
-----------------------------------------------------------------------------------------
| epoch   2 |  1000/ 2983 batches | lr 20.00 | ms/batch 14.70 | loss  5.43 | ppl   227.08
| epoch   2 |  2000/ 2983 batches | lr 20.00 | ms/batch 15.50 | loss  5.31 | ppl   202.46
-----------------------------------------------------------------------------------------
| end of epoch   2 | time: 46.72s | valid loss  5.29 | valid ppl   197.78
-----------------------------------------------------------------------------------------
| epoch   3 |  1000/ 2983 batches | lr 20.00 | ms/batch 14.21 | loss  5.11 | ppl   164.92
| epoch   3 |  2000/ 2983 batches | lr 20.00 | ms/batch 12.48 | loss  5.07 | ppl   159.30
-----------------------------------------------------------------------------------------
| end of epoch   3 | time: 42.86s | valid loss  5.16 | valid ppl   173.35
-----------------------------------------------------------------------------------------
| epoch   4 |  1000/ 2983 batches | lr 20.00 | ms/batch 14.82 | loss  4.93 | ppl   138.39
| epoch   4 |  2000/ 2983 batches | lr 20.00 | ms/batch 14.20 | loss  4.92 | ppl   137.51
-----------------------------------------------------------------------------------------
| end of epoch   4 | time: 45.11s | valid loss  5.08 | valid ppl   161.40
-----------------------------------------------------------------------------------------
| epoch   5 |  1000/ 2983 batches | lr 20.00 | ms/batch 14.48 | loss  4.81 | ppl   122.54
| epoch   5 |  2000/ 2983 batches | lr 20.00 | ms/batch 15.28 | loss  4.82 | ppl   123.86
-----------------------------------------------------------------------------------------
| end of epoch   5 | time: 46.12s | valid loss  5.03 | valid ppl   153.23
-----------------------------------------------------------------------------------------
| epoch   6 |  1000/ 2983 batches | lr 20.00 | ms/batch 13.98 | loss  4.72 | ppl   112.06
| epoch   6 |  2000/ 2983 batches | lr 20.00 | ms/batch 14.20 | loss  4.74 | ppl   114.29
-----------------------------------------------------------------------------------------
```