Sequence to Sequence Learning with Neural Networks_replication

September 16, 2023

nous allons construire un modèle d'apprentissage automatique pour passer d'une séquence à l'autre, en utilisant PyTorch et torchtext. Cela se fera sur les traductions de l'allemand vers l'anglais, mais les modèles peuvent être appliqués à tout problème impliquant de passer d'une séquence à une autre, comme le résumé, c'est-à-dire passer d'une séquence à une séquence plus courte dans la même langue.

Dans ce cahier, nous commencerons par comprendre les concepts généraux en implémentant le modèle de l'article Sequence to Sequence Learning with Neural Networks.

0.1 Préparation des données

Nous allons coder les modèles dans PyTorch et utiliser torchtext pour nous aider à faire tout le prétraitement requis. Nous utiliserons également spaCy pour aider à la tokenisation des données.

[ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim

from torchtext.datasets import Multi30k
from torchtext.data import Field, BucketIterator

import spacy
import numpy as np

import random
import math
import time
```

Nous allons définir les graines aléatoires pour des résultats déterministes.

[2]:
```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Ensuite, nous allons créer les tokenizers. Un tokenizer est utilisé pour transformer une chaîne contenant une phrase en une liste de jetons individuels qui composent cette chaîne, par exemple «

bonjour! » devient [“bon », « jour », « !”]. Nous allons commencer à parler des phrases comme étant une séquence de jetons à partir de maintenant, au lieu de dire qu’elles sont une séquence de mots. Quelle est la différence ? Eh bien, « bon » et « matin » sont à la fois des mots et des jetons, mais « ! » est un jeton, pas un mot.

spaCy a un modèle pour chaque langue (« de_core_news_sm » pour l’allemand et « en_core_web_sm » pour l’anglais) qui doivent être chargés afin que nous puissions accéder au tokenizer de chaque modèle.

Remarque : les modèles doivent d’abord être téléchargés à l’aide des éléments suivants sur la ligne de commande :

python -m spacy download en_core_web_sm python -m spacy download de_core_news_sm Nous chargeons les modèles comme tels:

[3]:
```
spacy_de = spacy.load('de_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')
```

Ensuite, nous créons les fonctions de tokenizer. Ceux-ci peuvent être transmis au texte torch et prendront la phrase comme une chaîne et retourneront la phrase comme une liste de jetons.

Dans le document que nous mettons en œuvre, ils trouvent bénéfique d’inverser l’ordre de l’entrée qui, selon eux, « introduit de nombreuses dépendances à court terme dans les données qui rendent le problème d’optimisation beaucoup plus facile ». Nous copions cela en inversant la phrase allemande après qu’elle ait été transformée en une liste de jetons.

[4]:
```
def tokenize_de(text):
    """
    Tokenizes German text from a string into a list of strings (tokens) and␣
 ↪reverses it
    """
    return [tok.text for tok in spacy_de.tokenizer(text)][::-1]

def tokenize_en(text):
    """
    Tokenizes English text from a string into a list of strings (tokens)
    """
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

orchText gère la façon dont les données doivent être traitées. Field

Nous définissons l’argument sur la fonction de tokenisation correcte pour chacun, l’allemand étant le champ (source) et l’anglais le champ (cible). Le champ ajoute également les jetons « début de séquence » et « fin de séquence » via les arguments et, et convertit tous les mots en minuscules.tokenizeSRCTRGinit_tokeneos_token

[5]:
```
SRC = Field(tokenize = tokenize_de,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)
```

```
TRG = Field(tokenize = tokenize_en,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)
```

```
/home/ben/miniconda3/envs/pytorch17/lib/python3.8/site-packages/torchtext-
0.9.0a0+c38fd42-py3.8-linux-x86_64.egg/torchtext/data/field.py:150: UserWarning:
Field class will be retired soon and moved to torchtext.legacy. Please see the
most recent release notes for further information.
  warnings.warn('{} class will be retired soon and moved to torchtext.legacy.
Please see the most recent release notes for further
information.'.format(self.__class__.__name__), UserWarning)
```

Ensuite, nous téléchargeons et chargeons les données du train, de la validation et des tests.

Le jeu de données que nous utiliserons est le jeu de données Multi30k. Il s'agit d'un ensemble de données avec ~30 000 phrases parallèles en anglais, allemand et français, chacune avec ~12 mots par phrase.

exts Spécifie les langues à utiliser comme source et cible (la source passe en premier) et spécifie le champ à utiliser pour la source et la cible.fields

[6]:
```
train_data, valid_data, test_data = Multi30k.splits(exts = ('.de', '.en'),
                                                    fields = (SRC, TRG))
```

```
/home/ben/miniconda3/envs/pytorch17/lib/python3.8/site-packages/torchtext-
0.9.0a0+c38fd42-py3.8-linux-x86_64.egg/torchtext/data/example.py:78:
UserWarning: Example class will be retired soon and moved to torchtext.legacy.
Please see the most recent release notes for further information.
  warnings.warn('Example class will be retired soon and moved to
torchtext.legacy. Please see the most recent release notes for further
information.', UserWarning)
```

Nous pouvons vérifier que nous avons chargé le bon nombre d'exemples:

[7]:
```
print(f"Number of training examples: {len(train_data.examples)}")
print(f"Number of validation examples: {len(valid_data.examples)}")
print(f"Number of testing examples: {len(test_data.examples)}")
```

```
Number of training examples: 29000
Number of validation examples: 1014
Number of testing examples: 1000
```

We can also print out an example, making sure the source sentence is reversed:

[8]:
```
print(vars(train_data.examples[0]))
```

```
{'src': ['.', 'büsche', 'vieler', 'nähe', 'der', 'in', 'freien', 'im', 'sind',
'männer', 'weiße', 'junge', 'zwei'], 'trg': ['two', 'young', ',', 'white',
'males', 'are', 'outside', 'near', 'many', 'bushes', '.']}
```

{'src': ['.', 'büsche', 'vieler', 'nähe', 'der', 'in', 'freien', 'im', 'sind', 'männer', 'weiße', 'junge', 'zwei'],

'trg': ['two', 'young', ',', 'white', 'males', 'are', 'outside', 'near', 'many', 'bushes', '.']} Le point est au début de la phrase allemande (src), il semble donc que la phrase ait été correctement inversée.

Ensuite, nous allons construire le vocabulaire pour les langues source et cible. Le vocabulaire est utilisé pour associer chaque jeton unique à un index (un entier). Les vocabulaires des langues source et cible sont distincts.

En utilisant l'argument, nous n'autorisons que les jetons qui apparaissent au moins 2 fois à apparaître dans notre vocabulaire. Les jetons qui n'apparaissent qu'une seule fois sont convertis en jeton (inconnu).min_freq

Il est important de noter que notre vocabulaire ne doit être construit qu'à partir de l'ensemble de formation et non de l'ensemble de validation/test. Cela empêche les « fuites d'informations » dans notre modèle, ce qui nous donne des scores de validation/test artificiellement gonflés.

[9]:
```
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)
```

[10]:
```
print(f"Unique tokens in source (de) vocabulary: {len(SRC.vocab)}")
print(f"Unique tokens in target (en) vocabulary: {len(TRG.vocab)}")
```

```
Unique tokens in source (de) vocabulary: 7853
Unique tokens in target (en) vocabulary: 5893
```

La dernière étape de la préparation des données consiste à créer les itérateurs. Ceux-ci peuvent être itérés pour renvoyer un lot de données qui aura un attribut (les tenseurs PyTorch contenant un lot de phrases sources numériques) et un attribut (les tenseurs PyTorch contenant un lot de phrases cibles numériques). Numérique est juste une façon sophistiquée de dire qu'ils ont été convertis d'une séquence de jetons lisibles en une séquence d'index correspondants, en utilisant le vocabulaire. srctrg

Nous devons également définir un fichier . Ceci est utilisé pour dire à torchText de mettre les tenseurs sur le GPU ou non. Nous utilisons la fonction, qui reviendra si un GPU est détecté sur notre ordinateur. Nous transmettons cela à l'itérateur.torch.devicetorch.cuda.is_available()Truedevice

Lorsque nous obtenons un lot d'exemples utilisant un itérateur, nous devons nous assurer que toutes les phrases sources sont complétées à la même longueur, la même chose que les phrases cibles. Heureusement, les itérateurs de torchText s'en chargent pour nous !

Nous utilisons un au lieu de la norme car il crée des lots de telle sorte qu'il minimise la quantité de remplissage dans les phrases source et cible. BucketIteratorIterator

[11]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[12]:
```
BATCH_SIZE = 128

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

```
/home/ben/miniconda3/envs/pytorch17/lib/python3.8/site-packages/torchtext-
0.9.0a0+c38fd42-py3.8-linux-x86_64.egg/torchtext/data/iterator.py:48:
UserWarning: BucketIterator class will be retired soon and moved to
torchtext.legacy. Please see the most recent release notes for further
information.
  warnings.warn('{} class will be retired soon and moved to torchtext.legacy.
Please see the most recent release notes for further
information.'.format(self.__class__.__name__), UserWarning)
```

0.2 Construction du Seq2Seq Model

Nous allons construire notre modèle en trois parties. L'encodeur, le décodeur et un modèle seq2seq qui encapsule l'encodeur et le décodeur et fournira un moyen d'interface avec chacun.

0.2.1 Encoder

Tout d'abord, l'encodeur, un LSTM à 2 couches. Le papier que nous mettons en œuvre utilise un LSTM à 4 couches, mais dans l'intérêt du temps de formation, nous l'avons réduit à 2 couches. Le concept de RNN multicouche est facile à étendre de 2 à 4 couches.

[13]:
```python
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.hid_dim = hid_dim
        self.n_layers = n_layers

        self.embedding = nn.Embedding(input_dim, emb_dim)

        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):

        #src = [src len, batch size]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src len, batch size, emb dim]

        outputs, (hidden, cell) = self.rnn(embedded)

        #outputs = [src len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer
```

```
        return hidden, cell
```

0.2.2 Decoder

Ensuite, nous allons construire notre décodeur, qui sera également un LSTM à 2 couches (4 dans le papier).

[14]:
```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.output_dim = output_dim
        self.hid_dim = hid_dim
        self.n_layers = n_layers

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)

        self.fc_out = nn.Linear(hid_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell):

        #input = [batch size]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #n directions in the decoder will both always be 1, therefore:
        #hidden = [n layers, batch size, hid dim]
        #context = [n layers, batch size, hid dim]

        input = input.unsqueeze(0)

        #input = [1, batch size]

        embedded = self.dropout(self.embedding(input))

        #embedded = [1, batch size, emb dim]

        output, (hidden, cell) = self.rnn(embedded, (hidden, cell))

        #output = [seq len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]
```

```
        #seq len and n directions will always be 1 in the decoder, therefore:
        #output = [1, batch size, hid dim]
        #hidden = [n layers, batch size, hid dim]
        #cell = [n layers, batch size, hid dim]

        prediction = self.fc_out(output.squeeze(0))

        #prediction = [batch size, output dim]

        return prediction, hidden, cell
```

0.2.3 Seq2Seq

Pour la dernière partie de la mise en œuvre, nous allons implémenter le modèle seq2seq. Cela permettra de gérer :

- réception de la phrase d'entrée/source
- Utilisation de l'encodeur pour produire les vecteurs de contexte
- Utilisation du décodeur pour produire la phrase de sortie/cible prévue

[15]:
```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

        assert encoder.hid_dim == decoder.hid_dim, \
            "Hidden dimensions of encoder and decoder must be equal!"
        assert encoder.n_layers == decoder.n_layers, \
            "Encoder and decoder must have equal number of layers!"

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use ground-truth inputs 75%␣
 ↪of the time

        batch_size = trg.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
```

```
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.
↪device)

        #last hidden state of the encoder is used as the initial hidden state␣
↪of the decoder
        hidden, cell = self.encoder(src)

        #first input to the decoder is the <sos> tokens
        input = trg[0,:]

        for t in range(1, trg_len):

            #insert input token embedding, previous hidden and previous cell␣
↪states
            #receive output tensor (predictions) and new hidden and cell states
            output, hidden, cell = self.decoder(input, hidden, cell)

            #place predictions in a tensor holding predictions for each token
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        return outputs
```

1 Entrainement du Seq2Seq Model

Maintenant que notre modèle est mis en œuvre, nous pouvons commencer à le former.

Tout d'abord, nous allons initialiser notre modèle. Comme mentionné précédemment, les dimensions d'entrée et de sortie sont définies par la taille du vocabulaire. Les dimesions d'incorporation et les abandons pour le codeur et le décodeur peuvent être différents, mais le nombre de couches et la taille des états masqués/cellulaires doivent être les mêmes.

Nous définissons ensuite l'encodeur, le décodeur puis notre modèle Seq2Seq, que nous plaçons sur le fichier .device

[16]:
```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
```

```
DEC_EMB_DIM = 256
HID_DIM = 512
N_LAYERS = 2
ENC_DROPOUT = 0.5
DEC_DROPOUT = 0.5

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM, N_LAYERS, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, HID_DIM, N_LAYERS, DEC_DROPOUT)

model = Seq2Seq(enc, dec, device).to(device)
```

La prochaine étape consiste à initialiser les poids de notre modèle. Dans l'article, ils indiquent qu'ils initialisent tous les poids à partir d'une distribution uniforme comprise entre -0,08 et +0,08, c'est-à-dire $(-0{,}08, 0{,}08)$.

Nous initialisons les poids dans PyTorch en créant une fonction que nous à notre modèle. Lors de l'utilisation de , la fonction sera appelée sur chaque module et sous-module de notre modèle. Pour chaque module, nous parcourons en boucle tous les paramètres et les échantillonnons à partir d'une distribution uniforme avec .applyapplyinit_weightsnn.init.uniform_

[17]:
```
def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.uniform_(param.data, -0.08, 0.08)

model.apply(init_weights)
```

[17]:
```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(7853, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.5)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(5893, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.5)
    (fc_out): Linear(in_features=512, out_features=5893, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

Nous définissons également une fonction qui calculera le nombre de paramètres pouvant être entraînés dans le modèle.

[18]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 13,898,501 trainable parameters
```

Nous définissons notre optimiseur, que nous utilisons pour mettre à jour nos paramètres dans la boucle d'entraînement.

[19]:
```
optimizer = optim.Adam(model.parameters())
```

Ensuite, nous définissons notre fonction de perte. La fonction calcule à la fois le log softmax et le log-vraisemblance négatif de nos prédictions. CrossEntropyLoss

Notre fonction de perte calcule la perte moyenne par jeton, mais en passant l'index du jeton comme argument, nous ignorons la perte chaque fois que le jeton cible est un jeton de rembourrage. ignore_index

[20]:
```
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

Ensuite, nous allons définir notre boucle d'entraînement.

Tout d'abord, nous allons mettre le modèle en « mode d'entraînement » avec . Cela activera l'abandon (et la normalisation par lots, que nous n'utilisons pas), puis itérera dans notre itérateur de données.model.train()

Enfin, nous retournons la perte qui est moyennée sur tous les lots.

[21]:
```
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in enumerate(iterator):

        src = batch.src
        trg = batch.trg

        optimizer.zero_grad()

        output = model(src, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]
```

```
        loss = criterion(output, trg)

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()

        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

Notre boucle d'évaluation est similaire à notre boucle d'entraînement, mais comme nous ne mettons à jour aucun paramètre, nous n'avons pas besoin de passer un optimiseur ou une valeur de clip.

N'oubliez pas de mettre le modèle en mode évaluation avec . Cela désactivera l'abandon (et la normalisation par lots, le cas échéant).model.eval()

Nous utilisons le bloc pour nous assurer qu'aucun dégradé n'est calculé dans le bloc. Cela réduit la consommation de mémoire et accélère les choses. with torch.no_grad()

La boucle d'itération est similaire (sans les mises à jour des paramètres), mais nous devons nous assurer que nous désactivons le forçage de l'enseignant pour l'évaluation. Cela entraînera le modèle à n'utiliser que ses propres prédictions pour faire d'autres prédictions dans une phrase, ce qui reflète la façon dont il serait utilisé dans le déploiement.

[22]:
```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in enumerate(iterator):

            src = batch.src
            trg = batch.trg

            output = model(src, trg, 0) #turn off teacher forcing

            #trg = [trg len, batch size]
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)
```

```
            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

Ensuite, nous allons créer une fonction que nous utiliserons pour nous dire combien de temps dure une époque

[23]:
```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Nous pouvons enfin commencer à former notre modèle!

À chaque époque, nous vérifierons si notre modèle a obtenu la meilleure perte de validation jusqu'à présent. Si c'est le cas, nous mettrons à jour notre meilleure perte de validation et enregistrerons les paramètres de notre modèle (appelé dans PyTorch). Ensuite, lorsque nous viendrons tester notre modèle, nous utiliserons les paramètres enregistrés utilisés pour obtenir la meilleure perte de validation. state_dict

Nous imprimerons à la fois la perte et la perplexité à chaque époque. Il est plus facile de voir un changement dans la perplexité qu'un changement dans la perte car les chiffres sont beaucoup plus importants.

[24]:
```
N_EPOCHS = 10
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut1-model.pt')
```

```
    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.
↪3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.
↪3f}')
```

```
/home/ben/miniconda3/envs/pytorch17/lib/python3.8/site-packages/torchtext-
0.9.0a0+c38fd42-py3.8-linux-x86_64.egg/torchtext/data/batch.py:23: UserWarning:
Batch class will be retired soon and moved to torchtext.legacy. Please see the
most recent release notes for further information.
  warnings.warn('{} class will be retired soon and moved to torchtext.legacy.
Please see the most recent release notes for further
information.'.format(self.__class__.__name__), UserWarning)
```

```
Epoch: 01 | Time: 0m 26s
        Train Loss: 5.052 | Train PPL: 156.386
         Val. Loss: 4.916 |  Val. PPL: 136.446
Epoch: 02 | Time: 0m 26s
        Train Loss: 4.483 | Train PPL:  88.521
         Val. Loss: 4.789 |  Val. PPL: 120.154
Epoch: 03 | Time: 0m 25s
        Train Loss: 4.195 | Train PPL:  66.363
         Val. Loss: 4.552 |  Val. PPL:  94.854
Epoch: 04 | Time: 0m 25s
        Train Loss: 3.963 | Train PPL:  52.625
         Val. Loss: 4.485 |  Val. PPL:  88.672
Epoch: 05 | Time: 0m 25s
        Train Loss: 3.783 | Train PPL:  43.955
         Val. Loss: 4.375 |  Val. PPL:  79.466
Epoch: 06 | Time: 0m 25s
        Train Loss: 3.636 | Train PPL:  37.957
         Val. Loss: 4.234 |  Val. PPL:  69.011
Epoch: 07 | Time: 0m 26s
        Train Loss: 3.506 | Train PPL:  33.329
         Val. Loss: 4.077 |  Val. PPL:  58.948
Epoch: 08 | Time: 0m 27s
        Train Loss: 3.370 | Train PPL:  29.090
         Val. Loss: 4.018 |  Val. PPL:  55.581
Epoch: 09 | Time: 0m 26s
        Train Loss: 3.241 | Train PPL:  25.569
         Val. Loss: 3.934 |  Val. PPL:  51.113
Epoch: 10 | Time: 0m 26s
        Train Loss: 3.157 | Train PPL:  23.492
         Val. Loss: 3.927 |  Val. PPL:  50.743
```

Nous allons charger les paramètres () qui ont donné à notre modèle la meilleure perte de validation et l'exécuter sur le jeu de test.state_dict

[25]:
```
model.load_state_dict(torch.load('tut1-model.pt'))

test_loss = evaluate(model, test_iterator, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

```
| Test Loss: 3.951 | Test PPL:  52.001 |
```

[ ]:
```

```