

# Rapport de progression

---

## Notre projet Application: PassWordGenius

---

### 1. Introduction

#### 1.1. Notre équipe:

Notre équipe se compose de:

- Olivier Fabre, uapv2014042
- François Demogue, uapv2101708

#### 1.2. Environnement

##### 1.2.1. Notre dépôt gitHub

Branches:

- main (master): <https://github.com/olfabre/amsProjetMaster1>
- olivier: <https://github.com/olfabre/amsProjetMaster1/tree/olivier>
- françois: <https://github.com/olfabre/amsProjetMaster1/tree/francois>

##### 1.2.2. Serveur Cuda (traitement GPU)

Serveur attaché au CERI avec un accès ssh: `ssh -p 22 uapvxxxxx@joyeux.univ-avignon.fr`

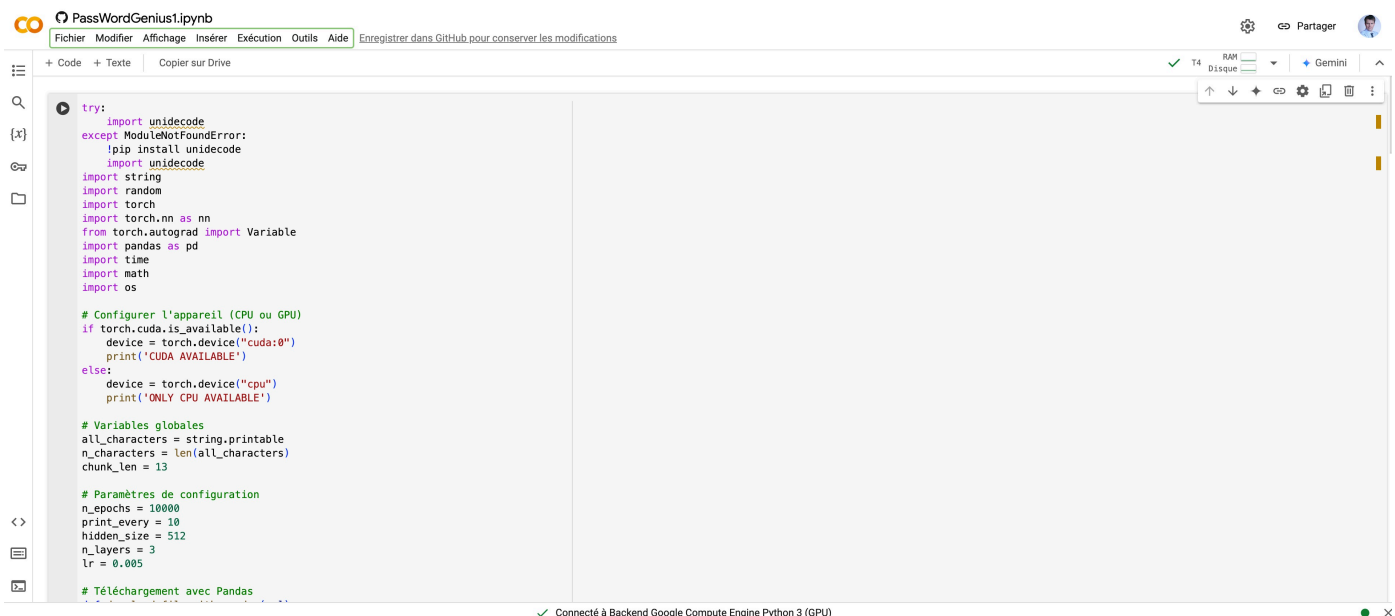
Activation avec la commande: `bash > conda activate shake`

**Note:** l'accès au serveur Cuda est très difficile et souvent inaccessible avec l'apparition de problèmes récurrents. Nous avons été obligés de changer de serveur en utilisant finalement Colab Google. Nous avons perdu tous nos fichiers et modèles car ils sont inaccessibles depuis deux semaines. On a été obligé de tout reprendre à zéro.

##### 1.2.3. Serveur Colab Google (traitement T4 GPU)

Service gratuit pour traiter nos données sur une architecture GPU relativement puissante.

Les codes ont été enregistrés sur notre dépôt: <https://github.com/olfabre/amsProjetMaster1/tree/olivier>



```
try:
    import unicode
except ModuleNotFoundError:
    !pip install unicode
import unicode
import string
import random
import torch
import torch.nn as nn
from torch.autograd import Variable
import pandas as pd
import time
import math
import os

# Configurer l'appareil (CPU ou GPU)
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print('CUDA AVAILABLE')
else:
    device = torch.device("cpu")
    print('ONLY CPU AVAILABLE')

# Variables globales
all_characters = string.printable
n_characters = len(all_characters)
chunk_len = 13

# Paramètres de configuration
n_epochs = 10000
print_every = 10
hidden_size = 512
n_layers = 3
lr = 0.005

# Téléchargement avec Pandas
```

### 1.2.4. Nos données

nous avons rassemblé toutes nos données et corpus d'entrée dans un même lieu à l'adresse suivante: <http://olivier-fabre.com/passwordgenius/>

Egalement en plus des corpus qui nous ont été remis, nous avons trouvé des corpus de mot de passe très intéressants classés par force à l'adresse suivante: <https://github.com/Infinitode/PWLDS>

Ce sera un set de data qui va contribuer à augmenter l'efficacité de notre application.

### 1.2.5. Le choix de PyTorch et de Python (version 3)

**PyTorch (syntaxe et simplicité)** : utilise une syntaxe qui ressemble beaucoup à Python standard, rendant le code plus lisible et plus proche de la programmation impérative. Cela rend la prise en main plus facile et le processus de développement plus fluide, surtout pour les débutants comme nous.

**PyTorch (débugage et flexibilité)** : grâce à sa nature impérative, il est facile de déboguer en utilisant des outils classiques comme `pdb` ou simplement en imprimant les valeurs des variables. C'est aussi plus flexible pour les expérimentations rapides ou les architectures de réseaux complexes.

**PyTorch (Gestion de l'Autograd)** : L'API **autograd** de PyTorch est intégrée et très intuitive pour les opérations de rétropropagation. Elle suit les opérations en direct, ce qui permet d'appliquer des gradients facilement, rendant la manipulation des réseaux plus naturelle.

PyTorch est souvent préféré par la communauté de recherche en intelligence artificielle et en apprentissage automatique, en particulier pour les projets de recherche académiques et expérimentaux. Les papiers de recherche et tutoriels sur les nouvelles architectures de modèles de réseaux de neurones sont souvent publiés avec du code PyTorch.

## 2. Travaux Pratiques

### 2.1. Atelier 1 - Corpus Shakespeare

Pour la génération de données séquentielles, il est nécessaire de disposer d'un ensemble de données permettant à l'IA d'apprendre à générer des éléments basiques successifs. Dans le contexte de la génération de textes, les éléments basiques constituant le texte sont les caractères. Ainsi, le modèle d'IA prendra en entrée successivement chacune des lettres (espaces et ponctuations compris) composant le texte et devra prédire le caractère suivant comme vu en cours. Ces éléments basiques (caractères dans notre cas) seront codés puis seront injectés dans le système IA pour prédire le caractère suivant. Dans le cadre de ce TP introductif, il sera alors nécessaire de :

- **Récupérer** et traiter les données
- **Apprendre** sur ces données (Un modèle RNN vous est fourni)
- **Evaluer** le système IA lors de la phase de génération de texte.

Le code initial joint

```
1  import unicodecode
2  import string
3  import random
4  import re
5
6  from os import listdir, path, makedirs, popen
7  from os.path import isdir, isfile, join
8
9  import torch
10 import torch.nn as nn
11 from torch.autograd import Variable
12
13 import time, math
14
15 import matplotlib.pyplot as plt
16 import matplotlib.ticker as ticker
17
18 from argparse import ArgumentParser
19
20 if torch.cuda.is_available():
21     device = torch.device("cuda:0")
22     print('CUDA AVAILABLE')
23 else:
24     device = torch.device("cpu")
25     print('ONLY CPU AVAILABLE')
26
27 all_characters = string.printable
28 n_characters = len(all_characters)
29 chunk_len = 13
30
31 n_epochs = 200000
```

```

32 print_every = 10
33 plot_every = 10
34 hidden_size = 512
35 n_layers = 3
36 lr = 0.005
37
38 def random_chunk(file):
39     start_index = random.randint(0, file_len - chunk_len)
40     end_index = start_index + chunk_len + 1
41     return file[start_index:end_index]
42
43 # Turn string into list of longs
44 def char_tensor(string):
45     tensor = torch.zeros(len(string)).long()
46     for c in range(len(string)):
47         tensor[c] = all_characters.index(string[c])
48     return Variable(tensor)
49
50 def random_training_set(file):
51     chunk = random_chunk(file)
52     inp = char_tensor(chunk[:-1]).to(device)
53     target = char_tensor(chunk[1:]).to(device)
54     return inp, target
55
56 def evaluate(decoder, prime_str='A', predict_len=100, temperature=0.8):
57     hidden = decoder.init_hidden()
58     prime_input = char_tensor(prime_str).to(device)
59     predicted = prime_str
60
61     # Use priming string to "build up" hidden state
62     for p in range(len(prime_str) - 1):
63         _, hidden = decoder(prime_input[p], hidden)
64     inp = prime_input[-1]
65
66     for p in range(predict_len):
67         output, hidden = decoder(inp, hidden)
68
69         # Sample from the network as a multinomial distribution
70         output_dist = output.data.view(-1).div(temperature).exp()
71         top_i = torch.multinomial(output_dist, 1)[0]
72
73         # Add predicted character to string and use as next input
74         predicted_char = all_characters[top_i]
75         predicted += predicted_char
76         inp = char_tensor(predicted_char).to(device)
77
78     return predicted
79
80 def time_since(since):

```

```

81     s = time.time() - since
82     m = math.floor(s / 60)
83     s -= m * 60
84     return '%dm %ds' % (m, s)
85
86 def train(inp, target):
87     hidden = decoder.init_hidden()
88     decoder.zero_grad()
89     loss = 0
90     for c in range(inp.size(0)): #range(chunk_len):
91         output, hidden = decoder(inp[c], hidden)
92         loss += criterion(output, target[c].unsqueeze(0))
93
94     loss.backward()
95     decoder_optimizer.step()
96
97     return loss.item() / chunk_len
98
99 class RNN(nn.Module):
100     def __init__(self, input_size, hidden_size, output_size, n_layers=1):
101         super(RNN, self).__init__()
102         self.input_size = input_size
103         self.hidden_size = hidden_size
104         self.output_size = output_size
105         self.n_layers = n_layers
106
107         self.encoder = nn.Embedding(input_size, hidden_size)
108         self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
109         self.decoder = nn.Linear(hidden_size, output_size)
110
111     def forward(self, input, hidden):
112         input = self.encoder(input.view(1, -1))
113         output, hidden = self.gru(input.view(1, 1, -1), hidden)
114         output = self.decoder(output.view(1, -1))
115         return output, hidden
116
117     def init_hidden(self):
118         return Variable(torch.zeros(self.n_layers, 1, self.hidden_size,
119 device=device))
120
121 def training(n_epochs, file):
122     print()
123     print('-----')
124     print('|  TRAIN  |')
125     print('-----')
126     print()
127
128     start = time.time()
129     all_losses = []

```

```

129     loss_avg = 0
130     best_loss = 100
131     print_every = n_epochs / 100
132
133     for epoch in range(1, n_epochs + 1):
134         loss = train(*random_training_set(file))
135         loss_avg += loss
136
137         if epoch % print_every == 0:
138             print('[%s (%d %d%%) %.4f (%.4f)]' % (time_since(start), epoch, epoch /
139 n_epochs * 100, loss_avg / epoch, loss))
140
141             if best_loss > (loss_avg / epoch):
142                 best_loss = loss_avg / epoch
143                 print('[%s (%d %d%%) %.4f (%.4f)]' % (time_since(start), epoch, epoch /
144 n_epochs * 100, loss_avg / epoch, loss))
145                 #print(evaluate('wh', 100), '\n')
146
147                 #if epoch % plot_every == 0:
148                     # all_losses.append(loss_avg / plot_every)
149                     # loss_avg = 0
150
151             #plt.figure()
152             #plt.plot(all_losses)
153
154 def evaluating(decoder, length):
155     print()
156     print('-----')
157     print('|   EVAL   |')
158     print('-----')
159     print()
160
161     try:
162         while True:
163             print('Enter a starting two or three characters')
164             input1 = input()
165             print()
166             if len(input1) > 0:
167                 print('Generated ', length, 'characters: ')
168                 print(evaluate(decoder = decoder, prime_str = input1, predict_len =
169 length, temperature = 0.8))
170             else:
171                 print(input1, ' length < 1')
172                 print('-----')
173                 print()
174
175     except KeyboardInterrupt:
176         print("Press Ctrl-C to terminate evaluating")
177         print('-----')

```

```

175
176 if __name__ == '__main__':
177
178     parser = ArgumentParser()
179     #
180     parser.add_argument("-d", "--trainingData", default="data/shakespeare.txt",
181 type=str, help="trainingData [path/to/the/data]")
182     parser.add_argument("-te", "--trainEval", default='train', type=str,
183 help="trainEval [train, eval]")
184     #
185     parser.add_argument("-r", "--run", default="rnnGeneration", type=str, help="name
186 of the model saved file")
187     parser.add_argument("-m", "--model", default='models', type=str, help="model to
188 save (train) or to load (eval) [path/to/the/model]")
189     #
190     parser.add_argument('--length', default=100, type=int, help="sequence length
191 during eval process [< 1000]")
192     parser.add_argument('--num_layers', default=2, type=int)
193     parser.add_argument('--hidden_size', default=128, type=int)
194     parser.add_argument('--max_epochs', default=10000, type=int)
195     #
196     args = parser.parse_args()
197     #
198     # repData = args.trainingData #"data/out/text10.txt"
199     repData = "shakespeare.txt"
200
201     file = unicode.unidecode(open(repData).read())
202     file_len = len(file)
203
204     decoder = RNN(n_characters, args.hidden_size, n_characters,
205 args.num_layers).to(device)
206     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
207     criterion = nn.CrossEntropyLoss()
208
209     n_epochs = args.max_epochs
210
211     print(random_chunk(file))
212
213     print('Training file_len =', file_len)
214
215     modelFile = args.run + "_" + str(args.num_layers) + "_" + str(args.hidden_size)
216 + ".pt"
217
218     if not path.exists(args.model):
219         makedirs(args.model)
220
221     if args.trainEval == 'train':
222         decoder.train()
223         training(n_epochs, file)

```

```

217     torch.save(decoder, join(args.model, modelFile))
218     elif args.trainEval == 'eval':
219         decoder = torch.load(join(args.model, modelFile))
220         decoder.eval().to(device)
221         evaluating(decoder, args.length)
222     else:
223         print('Choose trainEval option (--trainEval train/eval')
224
225
226

```

### Une explication détaillée de ce code:

Ce code est conçu pour entraîner un modèle de réseau de neurones récurrents (RNN) basé sur des **GRU** (Gated Recurrent Units). L'objectif est de prédire la prochaine lettre d'une séquence de texte, ici extraite d'un corpus de Shakespeare.

### Imports

```

1  import unicode
2  import string
3  import random
4  import re
5
6  from os import listdir, path, makedirs, popen
7  from os.path import isdir, isfile, join
8
9  import torch
10 import torch.nn as nn
11 from torch.autograd import variable
12
13 import time, math
14
15 import matplotlib.pyplot as plt
16 import matplotlib.ticker as ticker
17
18 from argparse import ArgumentParser
19

```

### Bibliothèques générales :

- `unicode` : Convertit le texte en ASCII pour éviter les caractères spéciaux.
- `string` : Fournit des outils pour manipuler des chaînes (comme `string.printable` pour les caractères imprimables).
- `random` : Génère des indices aléatoires pour sélectionner des morceaux de texte.
- `re` : Bibliothèque pour manipuler les expressions régulières.

### Modules système :



- Permettent la gestion des fichiers et des répertoires.

### PyTorch :

- `torch` et `torch.nn` : Implémentation du modèle et des optimisations.
- `variable` : Emballage des tenseurs pour les calculs différentiables (remplacé dans les nouvelles versions).

### Outils divers :

- `time` et `math` : Calculs et gestion du temps.
- `matplotlib` : Visualisation de la perte pendant l'entraînement.
- `argparse` : Permet de gérer les arguments en ligne de commande.

### Vérification du GPU

```
1 if torch.cuda.is_available():
2     device = torch.device("cuda:0")
3     print('CUDA AVAILABLE')
4 else:
5     device = torch.device("cpu")
6     print('ONLY CPU AVAILABLE')
```

Vérifie si un GPU est disponible. Si oui, le modèle s'exécute sur le GPU, sinon sur le CPU.

### Paramètres globaux

```
1 all_characters = string.printable
2 n_characters = len(all_characters)
3 chunk_len = 13
4
5 n_epochs = 200000
6 print_every = 10
7 plot_every = 10
8 hidden_size = 512
9 n_layers = 3
10 lr = 0.005
```

`all_characters` : Ensemble des caractères utilisés (lettres, chiffres, symboles).

`n_characters` : Nombre total de caractères possibles.

`chunk_len` : Longueur des morceaux de texte à traiter.

Hyperparamètres du modèle :

- `n_epochs` : Nombre total d'itérations d'entraînement.
- `hidden_size` : Taille des couches cachées.

- `n_layers`: Nombre de couches RNN.
- `lr`: Taux d'apprentissage.

### Fonction de préparation des données

```
1 def random_chunk(file):
2     start_index = random.randint(0, file_len - chunk_len)
3     end_index = start_index + chunk_len + 1
4     return file[start_index:end_index]
```

cette fonction sélectionne un morceau aléatoire du texte de longueur `chunk_len + 1` pour former une séquence d'entrée et une cible.

```
1 def char_tensor(string):
2     tensor = torch.zeros(len(string)).long()
3     for c in range(len(string)):
4         tensor[c] = all_characters.index(string[c])
5     return Variable(tensor)
```

cette fonction convertit une chaîne en tenseur où chaque caractère est représenté par son indice dans `all_characters`.

```
1 def random_training_set(file):
2     chunk = random_chunk(file)
3     inp = char_tensor(chunk[:-1]).to(device)
4     target = char_tensor(chunk[1:]).to(device)
5     return inp, target
```

cette fonction prépare un ensemble d'entraînement.

- `inp`: tout sauf le dernier caractère.
- `target`: tout sauf le premier caractère.

### Classe du modèle RNN

```

1 class RNN(nn.Module):
2     def __init__(self, input_size, hidden_size, output_size, n_layers=1):
3         super(RNN, self).__init__()
4         self.input_size = input_size
5         self.hidden_size = hidden_size
6         self.output_size = output_size
7         self.n_layers = n_layers
8
9         self.encoder = nn.Embedding(input_size, hidden_size)
10        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
11        self.decoder = nn.Linear(hidden_size, output_size)

```

### Initialisation :

- `nn.Embedding` : Transforme les indices des caractères en vecteurs.
- `nn.GRU` : Un GRU avec `hidden_size` neurones (nombre de neurones), `n_layers` couches (nombre de couches cachées).
- `nn.Linear` : Une couche entièrement connectée pour prédire le caractère suivant.

```

1 def forward(self, input, hidden):
2     input = self.encoder(input.view(1, -1))
3     output, hidden = self.gru(input.view(1, 1, -1), hidden)
4     output = self.decoder(output.view(1, -1))
5     return output, hidden
6
7 def init_hidden(self):
8     return Variable(torch.zeros(self.n_layers, 1, self.hidden_size, device=device))
9

```

### Propagation avant :

- Encode l'entrée, passe dans le GRU, puis dans la couche linéaire.
- Renvoie la sortie et l'état caché.

### Initialisation :

- Crée un tenseur d'état caché rempli de zéros.

### Fonction d'entraînement

```

1 def train(inp, target):
2     hidden = decoder.init_hidden()
3     decoder.zero_grad()
4     loss = 0
5     for c in range(inp.size(0)):
6         output, hidden = decoder(inp[c], hidden)
7         loss += criterion(output, target[c].unsqueeze(0))
8
9     loss.backward()
10    decoder_optimizer.step()
11
12    return loss.item() / chunk_len

```

### Étapes :

1. Initialise l'état caché.
2. Passe chaque caractère d'entrée dans le modèle.
3. Calcule la perte avec `CrossEntropyLoss`.
4. Effectue une rétropropagation pour mettre à jour les poids.

### Entraînement principal

```

1 def training(n_epochs, file):
2     for epoch in range(1, n_epochs + 1):
3         loss = train(*random_training_set(file))
4         loss_avg += loss
5         if epoch % print_every == 0:
6             print('[%s (%d %d%%) %.4f (%.4f)]' % (time_since(start), epoch, epoch /
n_epochs * 100, loss_avg / epoch, loss))

```

### Boucle d'entraînement :

- Exécute `train` sur des morceaux aléatoires.
- Affiche la perte moyenne toutes les `print_every` itérations.

### Évaluation

```

1 def evaluate(decoder, prime_str='A', predict_len=100, temperature=0.8):
2     hidden = decoder.init_hidden()
3     prime_input = char_tensor(prime_str).to(device)
4     predicted = prime_str
5
6     for p in range(len(prime_str) - 1):
7         _, hidden = decoder(prime_input[p], hidden)

```

```

8 inp = prime_input[-1]
9
10 for p in range(predict_len):
11     output, hidden = decoder(inp, hidden)
12     output_dist = output.data.view(-1).div(temperature).exp()
13     top_i = torch.multinomial(output_dist, 1)[0]
14     predicted_char = all_characters[top_i]
15     predicted += predicted_char
16     inp = char_tensor(predicted_char).to(device)
17
18 return predicted

```

Génère une séquence de texte en utilisant un modèle pré-entraîné.

`temperature` contrôle la diversité des sorties.

## Lancement

```

1 if __name__ == '__main__':
2     # Préparation du corpus
3     file = unicode.decode(open(repData).read())
4     file_len = len(file)
5
6     decoder = RNN(n_characters, args.hidden_size, n_characters,
7 args.num_layers).to(device)
8     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
9     criterion = nn.CrossEntropyLoss()
10
11 if args.trainEval == 'train':
12     training(n_epochs, file)
13 elif args.trainEval == 'eval':
14     decoder = torch.load(join(args.model, modelFile))
15     evaluating(decoder, args.length)

```

C'est la partie qui initialise le modèle et choisit entre l'entraînement ou l'évaluation.

Nous allons travailler sur Colab Google, qui permet de ne plus utiliser le serveur CERI.

Par conséquent nous sommes obligés d'adapter le code initial pour qu'il fonctionne sur cette plateforme

## Version initiale pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare\\_v1.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare_v1.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/shakespeare2.txt>

```

1 try:
2     import unicode
3 except ModuleNotFoundError:
4     !pip install unicode

```

```
5     import unicodecode
6 import string
7 import random
8 import re
9 import os
10 import requests
11
12 import torch
13 import torch.nn as nn
14 from torch.autograd import Variable
15
16 import time
17 import math
18 import matplotlib.pyplot as plt
19 from argparse import ArgumentParser
20
21 # Vérification du GPU
22 if torch.cuda.is_available():
23     device = torch.device("cuda:0")
24     print("CUDA AVAILABLE")
25 else:
26     device = torch.device("cpu")
27     print("ONLY CPU AVAILABLE")
28
29 # Paramètres globaux
30 all_characters = string.printable
31 n_characters = len(all_characters)
32 chunk_len = 13
33
34 n_epochs = 200000
35 print_every = 10
36 plot_every = 10
37 hidden_size = 512
38 n_layers = 3
39 lr = 0.005
40
41 # Téléchargement des données depuis une URL
42 def download_data(url, filename):
43     response = requests.get(url)
44     with open(filename, 'w', encoding='utf-8') as f:
45         f.write(response.text)
46
47 # Chargement des données
48 url = "https://olivier-fabre.com/passwordgenius/shakespeare2.txt"
49 data_dir = "data"
50 os.makedirs(data_dir, exist_ok=True)
51 data_path = os.path.join(data_dir, "shakespeare2.txt")
52
53 if not os.path.exists(data_path):
```

```

54     print("Téléchargement des données...")
55     download_data(url, data_path)
56
57 # Lecture et traitement du fichier
58 file = unicode.decode(open(data_path, "r", encoding="utf-8").read())
59 file_len = len(file)
60 print(f"Longueur du corpus : {file_len}")
61
62 # Fonctions de préparation des données
63 def random_chunk(file):
64     start_index = random.randint(0, file_len - chunk_len)
65     end_index = start_index + chunk_len + 1
66     return file[start_index:end_index]
67
68 def char_tensor(string):
69     tensor = torch.zeros(len(string)).long()
70     for c in range(len(string)):
71         tensor[c] = all_characters.index(string[c])
72     return Variable(tensor)
73
74 def random_training_set(file):
75     chunk = random_chunk(file)
76     inp = char_tensor(chunk[:-1]).to(device)
77     target = char_tensor(chunk[1:]).to(device)
78     return inp, target
79
80 # Définition du modèle
81 class RNN(nn.Module):
82     def __init__(self, input_size, hidden_size, output_size, n_layers=1):
83         super(RNN, self).__init__()
84         self.input_size = input_size
85         self.hidden_size = hidden_size
86         self.output_size = output_size
87         self.n_layers = n_layers
88
89         self.encoder = nn.Embedding(input_size, hidden_size)
90         self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
91         self.decoder = nn.Linear(hidden_size, output_size)
92
93     def forward(self, input, hidden):
94         input = self.encoder(input.view(1, -1))
95         output, hidden = self.gru(input.view(1, 1, -1), hidden)
96         output = self.decoder(output.view(1, -1))
97         return output, hidden
98
99     def init_hidden(self):
100         return Variable(torch.zeros(self.n_layers, 1, self.hidden_size,
101 device=device))

```

```

102 # Fonctions d'entraînement et d'évaluation
103 def train(inp, target):
104     hidden = decoder.init_hidden()
105     decoder.zero_grad()
106     loss = 0
107     for c in range(inp.size(0)):
108         output, hidden = decoder(inp[c], hidden)
109         loss += criterion(output, target[c].unsqueeze(0))
110     loss.backward()
111     decoder_optimizer.step()
112     return loss.item() / chunk_len
113
114 def training(n_epochs, file):
115     start = time.time()
116     for epoch in range(1, n_epochs + 1):
117         loss = train(*random_training_set(file))
118         if epoch % print_every == 0:
119             print(f"[{time_since(start)} ({epoch}/{n_epochs})] Perte :
120 {loss:.4f}")
121
122 def evaluate(decoder, prime_str="A", predict_len=100, temperature=0.8):
123     hidden = decoder.init_hidden()
124     prime_input = char_tensor(prime_str).to(device)
125     predicted = prime_str
126     for p in range(len(prime_str) - 1):
127         _, hidden = decoder(prime_input[p], hidden)
128     inp = prime_input[-1]
129     for p in range(predict_len):
130         output, hidden = decoder(inp, hidden)
131         output_dist = output.data.view(-1).div(temperature).exp()
132         top_i = torch.multinomial(output_dist, 1)[0]
133         predicted_char = all_characters[top_i]
134         predicted += predicted_char
135         inp = char_tensor(predicted_char).to(device)
136     return predicted
137
138 def time_since(since):
139     now = time.time()
140     s = now - since
141     m = math.floor(s / 60)
142     s -= m * 60
143     return f'{m}m {s:.2f}s'
144
145 # Lancement principal
146 if __name__ == "__main__":
147     decoder = RNN(n_characters, hidden_size, n_characters, n_layers).to(device)
148     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
149     criterion = nn.CrossEntropyLoss()

```



```

150     print("Début de l'entraînement...")
151     training(1000, file) # Ajustez n_epochs pour vos besoins
152
153     print("\nÉvaluation...")
154     print(evaluate(decoder, prime_str="To be or not to be", predict_len=200,
                    temperature=0.8))

```

## Version améliorée N°1 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare\\_v2.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare_v2.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/shakespeare2.txt>

Nous allons implémenter une nouvelle fonction `training`

## Caractéristiques :

### 1. Structure plus complexe :

- Utilise plusieurs chunks ( `chunk_count` ) pour calculer une moyenne des pertes.
- Effectue une évaluation intermédiaire à des intervalles réguliers ( `eval_every` ).

### 2. Avantages :

- Fournit une moyenne des pertes sur plusieurs chunks, ce qui est plus représentatif des performances globales.
- Introduit une évaluation intermédiaire pour vérifier les capacités du modèle à générer du texte.
- Suit la meilleure perte atteinte, ce qui permet un suivi des progrès.

### 3. Inconvénients :

- Plus lent car il traite plusieurs chunks à chaque itération.
- Code légèrement plus complexe à lire.

```

1  # Partie modifiée
2
3  def training(n_epochs, file, chunk_count=10):
4      print()
5      print('-----')
6      print('|  TRAIN  |')
7      print('-----')
8      print()
9
10     start = time.time()
11     all_losses = []
12     loss_avg = 0 # Moyenne des pertes sur tout l'entraînement
13     best_loss = float("inf")
14     print_every = n_epochs // 100
15     eval_every = n_epochs // 100
16
17     for epoch in range(1, n_epochs + 1):
18         losses = []

```

```

19     for _ in range(chunk_count):
20         loss = train(*random_training_set(file))
21         losses.append(loss)
22
23     # Moyenne sur les chunks
24     loss_avg += sum(losses) / chunk_count
25
26     if epoch % print_every == 0:
27         print('[%s (%d %d%%) Perte moyenne: %.4f Dernière perte: %.4f]' % (
28             time_since(start), epoch, epoch / n_epochs * 100, loss_avg / epoch,
29             losses[-1]))
30
31     if epoch % eval_every == 0:
32         print()
33         print(f"Évaluation à l'epoch {epoch}:")
34         print(evaluate(decoder, prime_str='wh', predict_len=100,
35             temperature=0.8))
36         print()
37
38     if best_loss > (loss_avg / epoch):
39         best_loss = loss_avg / epoch
40         print('[%s (%d %d%%) Nouvelle meilleure perte moyenne: %.4f]' % (
41             time_since(start), epoch, epoch / n_epochs * 100, best_loss))

```

## Version améliorée N°2 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare\\_v3.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare_v3.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/shakespeare2.txt>

Nous allons apporter des améliorations sur le code:

- Ajout de la sauvegarde du meilleur modèle dans `best_model.pth` si la perte moyenne s'améliore.
- `print_every` et `eval_every` peuvent être ajustés indépendamment.
- Plus d'informations sur les progrès de l'entraînement.

```

1  try:
2      import unicodecode
3  except ModuleNotFoundError:
4      !pip install unicodecode
5      import unicodecode
6  import string
7  import random
8  import re
9  import os
10 import requests
11
12 import torch

```

```
13 import torch.nn as nn
14 from torch.autograd import Variable
15
16 import time
17 import math
18 import matplotlib.pyplot as plt
19
20 # Vérification du GPU
21 if torch.cuda.is_available():
22     device = torch.device("cuda:0")
23     print("CUDA AVAILABLE")
24 else:
25     device = torch.device("cpu")
26     print("ONLY CPU AVAILABLE")
27
28 # Paramètres globaux
29 all_characters = string.printable
30 n_characters = len(all_characters)
31 chunk_len = 13
32
33 n_epochs = 200000
34 hidden_size = 512
35 n_layers = 3
36 lr = 0.005
37
38 # Fréquences d'affichage et d'évaluation
39 print_every_P = 50
40 eval_every_P = 100
41
42 # Téléchargement des données depuis une URL
43 def download_data(url, filename):
44     response = requests.get(url)
45     with open(filename, 'w', encoding='utf-8') as f:
46         f.write(response.text)
47
48 # Chargement des données
49 url = "https://olivier-fabre.com/passwordgenius/shakespeare2.txt"
50 data_dir = "data"
51 os.makedirs(data_dir, exist_ok=True)
52 data_path = os.path.join(data_dir, "shakespeare2.txt")
53
54 if not os.path.exists(data_path):
55     print("Téléchargement des données...")
56     download_data(url, data_path)
57
58 # Lecture et traitement du fichier
59 file = unicode.decode(open(data_path, "r", encoding="utf-8").read())
60 file_len = len(file)
61 print(f"Longueur du corpus : {file_len}")
```

```

62
63 # Fonctions de préparation des données
64 def random_chunk(file):
65     start_index = random.randint(0, file_len - chunk_len)
66     end_index = start_index + chunk_len + 1
67     return file[start_index:end_index]
68
69 def char_tensor(string):
70     tensor = torch.zeros(len(string)).long()
71     for c in range(len(string)):
72         tensor[c] = all_characters.index(string[c])
73     return Variable(tensor)
74
75 def random_training_set(file):
76     chunk = random_chunk(file)
77     inp = char_tensor(chunk[:-1]).to(device)
78     target = char_tensor(chunk[1:]).to(device)
79     return inp, target
80
81 # Définition du modèle
82 class RNN(nn.Module):
83     def __init__(self, input_size, hidden_size, output_size, n_layers=1):
84         super(RNN, self).__init__()
85         self.input_size = input_size
86         self.hidden_size = hidden_size
87         self.output_size = output_size
88         self.n_layers = n_layers
89
90         self.encoder = nn.Embedding(input_size, hidden_size)
91         self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
92         self.decoder = nn.Linear(hidden_size, output_size)
93
94     def forward(self, input, hidden):
95         input = self.encoder(input.view(1, -1))
96         output, hidden = self.gru(input.view(1, 1, -1), hidden)
97         output = self.decoder(output.view(1, -1))
98         return output, hidden
99
100     def init_hidden(self):
101         return Variable(torch.zeros(self.n_layers, 1, self.hidden_size,
102 device=device))
103
104 # Fonctions d'entraînement et d'évaluation
105 def train(inp, target):
106     hidden = decoder.init_hidden()
107     decoder.zero_grad()
108     loss = 0
109     for c in range(inp.size(0)):
110         output, hidden = decoder(inp[c], hidden)

```

```

110     loss += criterion(output, target[c].unsqueeze(0))
111     loss.backward()
112     decoder_optimizer.step()
113     return loss.item() / chunk_len
114
115 def training(n_epochs, file, chunk_count=10, print_every=10, eval_every=50):
116     print("\n-----")
117     print("|  TRAIN  |")
118     print("-----\n")
119
120     start = time.time()
121     all_losses = []
122     loss_avg = 0    # Moyenne des pertes sur tout l'entraînement
123     best_loss = float("inf")
124     model_save_path = os.path.join(data_dir, "best_model.pth")
125
126     for epoch in range(1, n_epochs + 1):
127         losses = []
128         for _ in range(chunk_count):
129             loss = train(*random_training_set(file))
130             losses.append(loss)
131
132         # Moyenne sur les chunks
133         loss_avg += sum(losses) / chunk_count
134
135         if epoch % print_every == 0:
136             print('[%s (%d/%d) Perte moyenne: %.4f Dernière perte: %.4f]' % (
137                 time_since(start), epoch, n_epochs, loss_avg / epoch, losses[-1]))
138
139         if epoch % eval_every == 0:
140             print(f"\nÉvaluation à l'epoch {epoch}:")
141             print(evaluate(decoder, prime_str='wh', predict_len=100,
142 temperature=0.8))
142             print()
143
144             # Sauvegarde du meilleur modèle
145             if best_loss > (loss_avg / epoch):
146                 best_loss = loss_avg / epoch
147                 torch.save(decoder.state_dict(), model_save_path)
148                 print('[%s (%d/%d) Nouvelle meilleure perte moyenne: %.4f Sauvegarde
149 du modèle.]' % (
150                     time_since(start), epoch, n_epochs, best_loss))
151
152 def evaluate(decoder, prime_str="A", predict_len=100, temperature=0.8):
153     hidden = decoder.init_hidden()
154     prime_input = char_tensor(prime_str).to(device)
155     predicted = prime_str
156     for p in range(len(prime_str) - 1):
157         _, hidden = decoder(prime_input[p], hidden)

```

```

157     inp = prime_input[-1]
158     for p in range(predict_len):
159         output, hidden = decoder(inp, hidden)
160         output_dist = output.data.view(-1).div(temperature).exp()
161         top_i = torch.multinomial(output_dist, 1)[0]
162         predicted_char = all_characters[top_i]
163         predicted += predicted_char
164         inp = char_tensor(predicted_char).to(device)
165     return predicted
166
167 def time_since(since):
168     now = time.time()
169     s = now - since
170     m = math.floor(s / 60)
171     s -= m * 60
172     return f'{m}m {s:.2f}s'
173
174 # Lancement principal
175 if __name__ == "__main__":
176     decoder = RNN(n_characters, hidden_size, n_characters, n_layers).to(device)
177     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
178     criterion = nn.CrossEntropyLoss()
179
180     print("Début de l'entraînement...")
181     training(1000, file, chunk_count=10, print_every=print_every_P,
182             eval_every=eval_every_P) # Ajustez n_epochs et les fréquences
183
184     print("\nÉvaluation finale...")
185     print(evaluate(decoder, prime_str="To be or not to be", predict_len=200,
186                   temperature=0.8))

```

Pour améliorer l'entraînement de votre modèle, nous pouvons ajuster les paramètres suivants :

## Hyperparamètres d'entraînement

Ces paramètres affectent directement l'apprentissage du modèle :

- **n\_epochs** :

Nous augmenterons le nombre d'époques si le modèle n'a pas assez de temps pour converger. Par exemple, nous pourrions prendre de 1000 à 5000 ou 10 000 époques ou 200 000 époques.

- **chunk\_count** :

Ce paramètre détermine combien de morceaux sont utilisés à chaque itération pour calculer la moyenne des pertes. Nous pourrions l'augmenter (par exemple, de 10 à 20) pour stabiliser les moyennes et améliorer la convergence.

- **lr (taux d'apprentissage)** :

Nous pourrions ajuster ce paramètre :

- Diminuons-le (par exemple, de 0.005 à 0.001) pour un apprentissage plus lent mais plus précis.
- Augmentons-le légèrement (par exemple, 0.01) si le modèle semble apprendre trop lentement.

## Hyperparamètres du modèle

Ces paramètres influencent la capacité du modèle à apprendre des données séquentielles :

- **hidden\_size** :  
La taille des couches cachées influence la complexité du modèle. Nous essayerons d'augmenter ce paramètre (par exemple, de 512 à 1024) pour permettre au modèle de capturer plus d'informations.
- **n\_layers** :  
Nous pourrions augmenter le nombre de couches (par exemple, de 3 à 4 ou 5) pour un modèle plus profond. Attention cependant que nous ne surchargeons pas notre GPU car il est quand même plutôt faible au regard de sa puissance.

## Fréquences d'affichage et d'évaluation

Ajustez ces paramètres pour suivre plus efficacement l'entraînement :

- **print\_every** :  
Nous pourrions réduire cet intervalle pour voir plus souvent la perte moyenne (par exemple, de 10 à 5).
- **eval\_every** :  
Nous pourrions évaluer le modèle plus fréquemment, en réduisant cet intervalle (par exemple, de 50 à 25).

## Fonction **evaluate**

### **predict\_len** :

Nous pourrions augmenter la longueur de prédiction pour mieux évaluer la capacité du modèle (par exemple, de 100 à 200).

### **temperature** :

Nous pourrions ajuster la diversité des prédictions :

- Une valeur plus basse (ex. 0.5) donne des sorties plus cohérentes mais moins variées.
- Une valeur plus élevée (ex. 1.2) produit des sorties plus imprévisibles.

## Taille du corpus

Ici nous avons utilisé très petit pour des tests. Nous pouvons utiliser un corpus plus grand ou plus diversifié si possible, tout en restant sous la limite de 10 Mo indiquée. Nous pourrions concaténer plusieurs œuvres pour enrichir l'apprentissage dans un futur.

## Régularisation

Nous pourrions ajouter des techniques pour éviter le surapprentissage :

- **Dropout** :  
Nous pourrions ajouter des couches de dropout pour régulariser le modèle :

```
1 | self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=0.2)
```

- **Gradient Clipping :**

Nous pourrions limiter la magnitude des gradients pour éviter des mises à jour trop importantes :

```
1 | torch.nn.utils.clip_grad_norm_(decoder.parameters(), max_norm=5)
```

## Optimiseur

Nous pourrions expérimenter avec d'autres optimiseurs, comme **AdamW** ou **RMSprop** :

```
1 | decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr)
```

## Plan d'amélioration

Voici un exemple de modifications progressives :

- Nous pouvons commencer par augmenter `n_epochs` et ajuster `lr`.
- Nous pourrions expérimenter avec `hidden_size` et `n_layers`.
- Nous pourrions intégrer des régularisations comme le dropout et le clipping.
- Nous pourrions tester avec des tailles de corpus plus grandes.

## Version améliorée N°3 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare\\_v4.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/ShakeSpeare_v4.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/shakespeare2.txt>

Nous allons apporter des améliorations sur le code:

### Changements apportés :

**Dropout** : Ajouté pour réduire le surapprentissage.

## Dropout

### Pourquoi ?

Le dropout est une technique de régularisation pour réduire le surapprentissage. Elle désactive aléatoirement un certain pourcentage de neurones pendant l'entraînement.

### Implémentation :

```
1 | self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=dropout)
```

- **Effet** : Chaque couche du GRU a une probabilité définie (`dropout_rate`) de désactiver des neurones à chaque étape.



- Paramètre ajouté : `dropout_rate = 0.2`.

**Gradient Clipping** : Stabilise l'entraînement en limitant les gradients.

## Gradient Clipping

### Pourquoi ?

Les gradients excessivement grands peuvent causer des mises à jour instables des poids, entraînant un échec de l'entraînement.

### Implémentation :

```
1 | torch.nn.utils.clip_grad_norm_(decoder.parameters(), max_norm=5)
```

- **Effet** : Limite la magnitude des gradients à `max_norm=5`. Cela évite les explosions de gradient dans les modèles récurrents comme le GRU.

**Optimiseur AdamW** : Utilisé pour de meilleures performances.

## Optimiseur AdamW

### Pourquoi ?

AdamW améliore Adam en ajoutant une régularisation explicite pour réduire l'effet des poids excessivement grands.

### Implémentation :

```
1 | decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr)
```

- **Effet** : Meilleure convergence et régularisation.

Paramètres ajustés:

- `n_epochs = 5000`, `hidden_size = 768`, `n_layers = 4`, `dropout_rate = 0.2`.
- `chunk_count` augmenté à 15 pour une moyenne plus fiable des pertes.
- **prime\_str** dans **evaluate** : Plus représentatif de Shakespeare.
- **Longueur de prédiction (predict\_len)** augmentée à 200 pour mieux évaluer la cohérence.

## **n\_epochs**

- **Avant** : 200 000.
- **Après** : 5 000 (valeur réduite pour un entraînement rapide).
- **Pourquoi ?** Les valeurs trop grandes peuvent entraîner un surapprentissage. Une valeur modérée avec des sauvegardes fréquentes est préférable.

## **hidden\_size**

- **Avant** : 512.
- **Après** : 768.
- **Pourquoi ?** Augmenter le nombre de neurones dans les couches cachées améliore la capacité du modèle à capturer des motifs complexes.

## **n\_layers**

- **Avant** : 3.
- **Après** : 4.
- **Pourquoi ?** Ajouter des couches rend le modèle plus profond, augmentant sa capacité d'abstraction.

## **lr (Learning Rate)**

- **Avant** : 0.005.
- **Après** : 0.002.
- **Pourquoi ?** Un taux d'apprentissage plus faible permet des mises à jour plus stables.

## Longueur de prédiction et chaîne d'amorçage ( **evaluate** )

### **Pourquoi ?**

Ces modifications permettent de mieux évaluer la capacité du modèle à générer du texte :

#### a. **predict\_len**

- **Avant** : 100.
- **Après** : 200.
- **Effet** : Génère des séquences plus longues pour observer la cohérence.

## b. `prime_str`

- **Avant :** `"wh"`.
- **Après :** `"To be or not to be"`.
- **Effet :** Utiliser une phrase représentative du corpus (Shakespeare) donne des résultats plus significatifs.

## Sauvegarde du modèle

### Pourquoi ?

Sauvegarder le meilleur modèle basé sur la perte moyenne garantit de ne pas perdre les progrès réalisés pendant l'entraînement.

### Implémentation :

```
1 if best_loss > (loss_avg / epoch):
2     best_loss = loss_avg / epoch
3     torch.save(decoder.state_dict(), model_save_path)
```

- **Effet :** Le modèle avec la meilleure perte est sauvegardé dans `best_model.pth`.

## Visualisation des pertes

### Pourquoi ?

Les pertes moyennes et individuelles sont imprimées pour suivre la progression de l'entraînement.

### Implémentation :

```
1 if epoch % print_every == 0:
2     print('[%s (%d/%d) Perte moyenne: %.4f Dernière perte: %.4f]' % (
3         time_since(start), epoch, n_epochs, loss_avg / epoch, losses[-1]))
```

```
1 try:
2     import unicode
3 except ModuleNotFoundError:
4     !pip install unicode
5     import unicode
6 import string
7 import random
8 import re
```

```
9 import os
10 import requests
11
12 import torch
13 import torch.nn as nn
14 from torch.autograd import Variable
15
16 import time
17 import math
18 import matplotlib.pyplot as plt
19
20 # Vérification du GPU
21 if torch.cuda.is_available():
22     device = torch.device("cuda:0")
23     print("CUDA AVAILABLE")
24 else:
25     device = torch.device("cpu")
26     print("ONLY CPU AVAILABLE")
27
28 # Paramètres globaux
29 all_characters = string.printable
30 n_characters = len(all_characters)
31 chunk_len = 13
32
33 # Paramètres modifiables
34 n_epochs = 5000
35 hidden_size = 768
36 n_layers = 4
37 lr = 0.002
38 dropout_rate = 0.2 # Ajout du dropout
39
40 # Téléchargement des données depuis une URL
41 def download_data(url, filename):
42     response = requests.get(url)
43     with open(filename, 'w', encoding='utf-8') as f:
44         f.write(response.text)
45
46 # Chargement des données
47 url = "https://olivier-fabre.com/passwordgenius/shakespeare2.txt"
48 data_dir = "data"
49 os.makedirs(data_dir, exist_ok=True)
50 data_path = os.path.join(data_dir, "shakespeare2.txt")
51
52 if not os.path.exists(data_path):
53     print("Téléchargement des données...")
54     download_data(url, data_path)
55
56 # Lecture et traitement du fichier
57 file = unicode.unidecode(open(data_path, "r", encoding="utf-8").read())
```

```

58 file_len = len(file)
59 print(f"Longueur du corpus : {file_len}")
60
61 # Fonctions de préparation des données
62 def random_chunk(file):
63     start_index = random.randint(0, file_len - chunk_len)
64     end_index = start_index + chunk_len + 1
65     return file[start_index:end_index]
66
67 def char_tensor(string):
68     tensor = torch.zeros(len(string)).long()
69     for c in range(len(string)):
70         tensor[c] = all_characters.index(string[c])
71     return Variable(tensor)
72
73 def random_training_set(file):
74     chunk = random_chunk(file)
75     inp = char_tensor(chunk[:-1]).to(device)
76     target = char_tensor(chunk[1:]).to(device)
77     return inp, target
78
79 # Définition du modèle
80 class RNN(nn.Module):
81     def __init__(self, input_size, hidden_size, output_size, n_layers=1,
82 dropout=0.0):
83         super(RNN, self).__init__()
84         self.input_size = input_size
85         self.hidden_size = hidden_size
86         self.output_size = output_size
87         self.n_layers = n_layers
88
89         self.encoder = nn.Embedding(input_size, hidden_size)
90         self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=dropout)
91         self.decoder = nn.Linear(hidden_size, output_size)
92
93     def forward(self, input, hidden):
94         input = self.encoder(input.view(1, -1))
95         output, hidden = self.gru(input.view(1, 1, -1), hidden)
96         output = self.decoder(output.view(1, -1))
97         return output, hidden
98
99     def init_hidden(self):
100         return Variable(torch.zeros(self.n_layers, 1, self.hidden_size,
101 device=device))
102
103 # Fonctions d'entraînement et d'évaluation
104 def train(inp, target):
105     hidden = decoder.init_hidden()
106     decoder.zero_grad()

```

```

105     loss = 0
106     for c in range(inp.size(0)):
107         output, hidden = decoder(inp[c], hidden)
108         loss += criterion(output, target[c].unsqueeze(0))
109     loss.backward()
110     torch.nn.utils.clip_grad_norm_(decoder.parameters(), max_norm=5) # Gradient
Clipping
111     decoder_optimizer.step()
112     return loss.item() / chunk_len
113
114 def training(n_epochs, file, chunk_count=10, print_every=10, eval_every=50):
115     print("\n-----")
116     print("|  TRAIN  |")
117     print("-----\n")
118
119     start = time.time()
120     all_losses = []
121     loss_avg = 0 # Moyenne des pertes sur tout l'entraînement
122     best_loss = float("inf")
123     model_save_path = os.path.join(data_dir, "best_model.pth")
124
125     for epoch in range(1, n_epochs + 1):
126         losses = []
127         for _ in range(chunk_count):
128             loss = train(*random_training_set(file))
129             losses.append(loss)
130
131         # Moyenne sur les chunks
132         loss_avg += sum(losses) / chunk_count
133
134         if epoch % print_every == 0:
135             print('[%s (%d/%d) Perte moyenne: %.4f Dernière perte: %.4f]' % (
136                 time_since(start), epoch, n_epochs, loss_avg / epoch, losses[-1]))
137
138         if epoch % eval_every == 0:
139             print(f"\névaluation à l'epoch {epoch}:")
140             print(evaluate(decoder, prime_str='To be', predict_len=200,
temperature=0.8))
141             print()
142
143         # Sauvegarde du meilleur modèle
144         if best_loss > (loss_avg / epoch):
145             best_loss = loss_avg / epoch
146             torch.save(decoder.state_dict(), model_save_path)
147             print('[%s (%d/%d) Nouvelle meilleure perte moyenne: %.4f Sauvegarde
du modèle.]' % (
148                 time_since(start), epoch, n_epochs, best_loss))
149
150 def evaluate(decoder, prime_str="To be", predict_len=200, temperature=0.8):

```

```

151     hidden = decoder.init_hidden()
152     prime_input = char_tensor(prime_str).to(device)
153     predicted = prime_str
154     for p in range(len(prime_str) - 1):
155         _, hidden = decoder(prime_input[p], hidden)
156     inp = prime_input[-1]
157     for p in range(predict_len):
158         output, hidden = decoder(inp, hidden)
159         output_dist = output.data.view(-1).div(temperature).exp()
160         top_i = torch.multinomial(output_dist, 1)[0]
161         predicted_char = all_characters[top_i]
162         predicted += predicted_char
163         inp = char_tensor(predicted_char).to(device)
164     return predicted
165
166 def time_since(since):
167     now = time.time()
168     s = now - since
169     m = math.floor(s / 60)
170     s -= m * 60
171     return f'{m}m {s:.2f}s'
172
173 # Lancement principal
174 if __name__ == "__main__":
175     decoder = RNN(n_characters, hidden_size, n_characters, n_layers,
176 dropout=dropout_rate).to(device)
177     decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr)
178     criterion = nn.CrossEntropyLoss()
179
180     print("Début de l'entraînement...")
181     training(n_epochs, file, chunk_count=15, print_every=50, eval_every=100)
182
183     print("\nÉvaluation finale...")
184     print(evaluate(decoder, prime_str="To be or not to be", predict_len=200,
185 temperature=0.8))

```

Pour continuer à entraîner notre modèle sauvegardé dans `data/best_model.pth`

## Recharger un modèle sauvegardé

Pour réutiliser un modèle sauvegardé plus tard :

### 1. Charger l'état du modèle :

```

1 decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
2 decoder.load_state_dict(torch.load("data/best_model.pth"))
3 decoder.eval() # Important pour désactiver le mode entraînement

```

### 2. Vérifiez qu'il est correctement chargé :

- Testez la génération avec le modèle chargé :

```
1 | print(sample(decoder, "A"))
```

### 3. Travaux Pratiques (suite)

#### 2.1. Atelier 2 - Corpus de prénoms (Russes)

Ce programme (python) permet d'apprendre depuis un ensemble de prénoms pour la génération de prénoms. L'algorithme d'apprentissage est un réseau de neurones récurrent (RNN) qui prend en entrée un caractère à la fois et essaie de prédire le suivant.

Lors de la première phase, il faut lui fournir donc des prénoms de textes dans un seul fichier pour le volet apprentissage. Un fichier contenant un ensemble de prénoms russes (contenant le plus grand nombre de prénoms) est donné dans la section E-UAPV du défi. D'autres langues sont disponibles ici : <https://download.pytorch.org/tutorial/data.zip>.

Nous avons adapté le code pour le faire tourner dans Colab Google

#### Version initiale N°1 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V1.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V1.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

```
1 | try:
2 |     import unicodecode
3 | except ModuleNotFoundError:
4 |     !pip install unicodecode
5 |     import unicodecode
6 |
7 | import requests
8 | import torch
9 | import torch.nn as nn
10 | from torch.autograd import variable
11 | import time
12 | import math
13 | import string
14 | import random
15 | import os
16 |
17 | # Vérification GPU
18 | device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
19 | print(f"Device utilisé: {device}")
20 |
21 | # Téléchargement des données
```



```

22 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
23 data_dir = "data"
24 os.makedirs(data_dir, exist_ok=True)
25 data_path = os.path.join(data_dir, "russian.txt")
26
27 if not os.path.exists(data_path):
28     print("Chargement des données en cours...")
29     response = requests.get(url)
30     with open(data_path, 'w', encoding='utf-8') as f:
31         f.write(response.text)
32
33 # Chargement des données
34 def unicode_to_ascii(s):
35     return ''.join(
36         c for c in unidecode.unidecode(s)
37         if c in (string.ascii_letters + " .,;'-")
38     )
39
40 def read_lines(filename):
41     with open(filename, encoding='utf-8') as f:
42         return [unicode_to_ascii(line.strip().lower()) for line in f]
43
44 lines = read_lines(data_path)
45 print(f"Nombre de noms: {len(lines)}")
46
47 # Paramètres globaux
48 all_letters = string.ascii_letters + " .,;'-"
49 n_letters = len(all_letters) + 1 # EOS marker
50 hidden_size = 128
51 n_layers = 2
52 lr = 0.005
53 bidirectional = True
54 max_length = 20
55
56 # Fonctions utilitaires
57 def char_tensor(string):
58     tensor = torch.zeros(len(string)).long()
59     for c in range(len(string)):
60         tensor[c] = all_letters.index(string[c])
61     return tensor
62
63 def input_tensor(line):
64     tensor = torch.zeros(len(line), 1, n_letters)
65     for li in range(len(line)):
66         letter = line[li]
67         tensor[li][0][all_letters.find(letter)] = 1
68     return tensor
69
70 def target_tensor(line):

```

```

71     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
72     letter_indexes.append(n_letters - 1) # EOS
73     return torch.LongTensor(letter_indexes)
74
75 def random_training_example(lines):
76     line = random.choice(lines)
77     input_line_tensor = input_tensor(line)
78     target_line_tensor = target_tensor(line)
79     return input_line_tensor, target_line_tensor
80
81 # Définition du modèle
82 class RNNLight(nn.Module):
83     def __init__(self, input_size, hidden_size, output_size):
84         super(RNNLight, self).__init__()
85         self.input_size = input_size
86         self.hidden_size = hidden_size
87         self.output_size = output_size
88         self.bidirectional = bidirectional
89         self.num_directions = 2 if self.bidirectional else 1
90         self.rnn = nn.RNN(
91             input_size=input_size, hidden_size=hidden_size,
92             num_layers=1, bidirectional=self.bidirectional, batch_first=True
93         )
94         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
95         self.dropout = nn.Dropout(0.1)
96         self.softmax = nn.LogSoftmax(dim=1)
97
98     def forward(self, input, hidden):
99         _, hidden = self.rnn(input.unsqueeze(0), hidden)
100         hidden_concat = hidden if not self.bidirectional else
torch.cat((hidden[0], hidden[1]), 1)
101         output = self.out(hidden_concat)
102         output = self.dropout(output)
103         return self.softmax(output), hidden
104
105     def init_hidden(self):
106         return torch.zeros(self.num_directions, 1, self.hidden_size,
device=device)
107
108 # Entraînement
109 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
criterion):
110     target_line_tensor.unsqueeze_(-1)
111     hidden = decoder.init_hidden()
112     decoder.zero_grad()
113     loss = 0
114     for i in range(input_line_tensor.size(0)):
115         output, hidden = decoder(input_line_tensor[i].to(device),
hidden.to(device))

```

```

116         l = criterion(output.to(device), target_line_tensor[i].to(device))
117         loss += l
118     loss.backward()
119     decoder_optimizer.step()
120     return loss.item() / input_line_tensor.size(0)
121
122 def training(n_epochs, lines, decoder, decoder_optimizer, criterion):
123     print("\n-----\n|  Entrainement  |\n-----\n")
124     start = time.time()
125     total_loss = 0
126     for epoch in range(1, n_epochs + 1):
127         input_line_tensor, target_line_tensor = random_training_example(lines)
128         loss = train(input_line_tensor, target_line_tensor, decoder,
129 decoder_optimizer, criterion)
129         total_loss += loss
130         if epoch % 500 == 0:
131             print(f"{time_since(start)} ({epoch}/{n_epochs}) Perte: {total_loss /
132 epoch:.4f}")
133
134 # Génération de noms
135 def sample(decoder, start_letter='A'):
136     with torch.no_grad():
137         hidden = decoder.init_hidden()
138         input = input_tensor(start_letter)
139         output_name = start_letter
140         for _ in range(max_length):
141             output, hidden = decoder(input[0].to(device), hidden.to(device))
142             topi = output.topk(1)[1][0][0]
143             if topi == n_letters - 1:
144                 break
145             else:
146                 letter = all_letters[topi]
147                 output_name += letter
148                 input = input_tensor(letter)
149         return output_name
150
151 def time_since(since):
152     """Retourne le temps écoulé au format mm:ss"""
153     now = time.time()
154     s = now - since
155     m = math.floor(s / 60)
156     s -= m * 60
157     return f"{m}m {s:.2f}s"
158
159 # Exécution principale
160 if __name__ == "__main__":
161     decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
162     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
163     criterion = nn.CrossEntropyLoss()

```

```

163     n_epochs = 5000
164
165     print("Demarrage entraînement...")
166     training(n_epochs, lines, decoder, decoder_optimizer, criterion)
167
168     print("\nGénération de noms:")
169     for letter in "ABC":
170         print(sample(decoder, letter))
171

```

## Version améliorée N°2 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V2.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V2.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

### Résumé des améliorations

1. **Sauvegarde automatique** : Le modèle avec la meilleure perte de validation est sauvegardé dans `best_model_generation_prenom.pth`.
2. **Division des données** : Les données sont divisées en 80% (entraînement), 10% (validation), et 10% (test).
3. Progrès affichés:
  - Affichage de la perte d'entraînement et de validation.
- Sauvegarde du modèle lorsqu'une meilleure perte de validation est atteinte.

```

1  try:
2      import unicode
3  except ModuleNotFoundError:
4      !pip install unicode
5      import unicode
6
7  import requests
8  import torch
9  import torch.nn as nn
10 from torch.autograd import variable
11 import time
12 import math
13 import string
14 import random
15 import os
16
17 # Vérification GPU
18 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

```

```

19 print(f"Appareil utilisé : {device}")
20
21 # Téléchargement des données
22 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
23 data_dir = "data"
24 os.makedirs(data_dir, exist_ok=True)
25 data_path = os.path.join(data_dir, "russian.txt")
26
27 if not os.path.exists(data_path):
28     print("Téléchargement des données...")
29     response = requests.get(url)
30     with open(data_path, 'w', encoding='utf-8') as f:
31         f.write(response.text)
32
33 # Chargement des données
34 def unicode_to_ascii(s):
35     return ''.join(
36         c for c in unidecode.unidecode(s)
37         if c in (string.ascii_letters + " .,;'-")
38     )
39
40 def read_lines(filename):
41     with open(filename, encoding='utf-8') as f:
42         return [unicode_to_ascii(line.strip().lower()) for line in f]
43
44 lines = read_lines(data_path)
45 print(f"Nombre de prénoms : {len(lines)}")
46
47 # Division des données
48 random.shuffle(lines)
49 train_split = int(0.8 * len(lines))
50 valid_split = int(0.1 * len(lines))
51 train_lines = lines[:train_split]
52 valid_lines = lines[train_split:train_split + valid_split]
53 test_lines = lines[train_split + valid_split:]
54 print(f"Ensemble d'entraînement : {len(train_lines)}, Validation : {len(valid_lines)}, Test : {len(test_lines)}")
55
56 # Paramètres globaux
57 all_letters = string.ascii_letters + " .,;'-"
58 n_letters = len(all_letters) + 1 # EOS marker
59 hidden_size = 128
60 n_layers = 2
61 lr = 0.005
62 bidirectional = True
63 max_length = 20
64
65 # Fonctions utilitaires
66 def char_tensor(string):

```

```

67     tensor = torch.zeros(len(string)).long()
68     for c in range(len(string)):
69         tensor[c] = all_letters.index(string[c])
70     return tensor
71
72 def input_tensor(line):
73     tensor = torch.zeros(len(line), 1, n_letters)
74     for li in range(len(line)):
75         letter = line[li]
76         tensor[li][0][all_letters.find(letter)] = 1
77     return tensor
78
79 def target_tensor(line):
80     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
81     letter_indexes.append(n_letters - 1) # EOS
82     return torch.LongTensor(letter_indexes)
83
84 def random_training_example(lines):
85     line = random.choice(lines)
86     input_line_tensor = input_tensor(line)
87     target_line_tensor = target_tensor(line)
88     return input_line_tensor, target_line_tensor
89
90 # Fonction pour afficher le temps écoulé
91 def time_since(since):
92     now = time.time()
93     s = now - since
94     m = math.floor(s / 60)
95     s -= m * 60
96     return f"{m}m {s:.2f}s"
97
98 # Définition du modèle
99 class RNNLight(nn.Module):
100     def __init__(self, input_size, hidden_size, output_size):
101         super(RNNLight, self).__init__()
102         self.input_size = input_size
103         self.hidden_size = hidden_size
104         self.output_size = output_size
105         self.bidirectional = bidirectional
106         self.num_directions = 2 if self.bidirectional else 1
107         self.rnn = nn.RNN(
108             input_size=input_size, hidden_size=hidden_size,
109             num_layers=1, bidirectional=self.bidirectional, batch_first=True
110         )
111         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
112         self.dropout = nn.Dropout(0.1)
113         self.softmax = nn.LogSoftmax(dim=1)
114
115     def forward(self, input, hidden):

```

```

116         _, hidden = self.rnn(input.unsqueeze(0), hidden)
117         hidden_concat = hidden if not self.bidirectional else
torch.cat((hidden[0], hidden[1]), 1)
118         output = self.out(hidden_concat)
119         output = self.dropout(output)
120         return self.softmax(output), hidden
121
122     def init_hidden(self):
123         return torch.zeros(self.num_directions, 1, self.hidden_size,
device=device)
124
125     # Entraînement avec sauvegarde
126     def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
criterion):
127         target_line_tensor = target_line_tensor.to(device) # Déplacement vers le bon
dispositif
128         hidden = decoder.init_hidden().to(device) # Initialisation sur le bon
dispositif
129         decoder.zero_grad()
130         loss = 0
131         for i in range(input_line_tensor.size(0)):
132             input_tensor = input_line_tensor[i].to(device) # Déplacement explicite
133             target_tensor = target_line_tensor[i].unsqueeze(0).to(device) #
Déplacement explicite
134             output, hidden = decoder(input_tensor, hidden.detach()) # Utilisation de
detach
135             l = criterion(output, target_tensor)
136             loss += l
137             loss.backward()
138             decoder_optimizer.step()
139             return loss.item() / input_line_tensor.size(0)
140
141     def validation(input_line_tensor, target_line_tensor, decoder, criterion):
142         with torch.no_grad(): # Pas de calcul de gradients pendant la validation
143             target_line_tensor = target_line_tensor.to(device)
144             hidden = decoder.init_hidden().to(device)
145             loss = 0
146             for i in range(input_line_tensor.size(0)):
147                 input_tensor = input_line_tensor[i].to(device)
148                 target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
149                 output, hidden = decoder(input_tensor, hidden.detach())
150                 l = criterion(output, target_tensor)
151                 loss += l
152             return loss.item() / input_line_tensor.size(0)
153
154     def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion):
155         print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
156         start = time.time()

```

```

157     best_loss = float("inf")
158     model_path = "best_model_generation_prenom.pth"
159
160     for epoch in range(1, n_epochs + 1):
161         # Entraînement
162         input_line_tensor, target_line_tensor =
random_training_example(train_lines)
163         train_loss = train(input_line_tensor, target_line_tensor, decoder,
decoder_optimizer, criterion)
164
165         # Validation
166         input_line_tensor, target_line_tensor =
random_training_example(valid_lines)
167         val_loss = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
168
169         # Sauvegarde du meilleur modèle
170         if val_loss < best_loss:
171             best_loss = val_loss
172             torch.save(decoder.state_dict(), model_path)
173             print(f"Époch {epoch} : La perte de validation a diminué à
{best_loss:.4f}. Modèle sauvegardé.")
174
175         if epoch % 500 == 0:
176             print(f"{time_since(start)} Époch {epoch}/{n_epochs}, Perte
entraînement : {train_loss:.4f}, Perte validation : {val_loss:.4f}")
177
178     # Exécution principale
179     if __name__ == "__main__":
180         decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
181         decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
182         criterion = nn.CrossEntropyLoss()
183         n_epochs = 5000
184
185         print("Démarrage de l'entraînement...")
186         training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
187
188         print("\nGénération de prénoms :")
189         for letter in "ABC":
190             print(sample(decoder, letter))
191

```

### Version améliorée N°3 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V3.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V3.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>



Voici une version mise à jour du code, nous incluons :

- Une augmentation du nombre d'époques : nous pouvons maintenant ajuster facilement le nombre d'époques avec un affichage clair des progrès.
- Nous souhaitons un affichage d'une série de prénoms générés : à chaque amélioration du modèle (meilleure perte de validation), une série de prénoms est générée et affichée.

### Augmentation des époques :

- Le nombre d'époques a été augmenté :

```
1 | n_epochs = 10000 # Augmentez le nombre d'époques ici
```

### Génération de prénoms après chaque amélioration du modèle :

- La fonction `generate_series` génère des prénoms à partir d'une série de lettres de départ.
- Elle est appelée chaque fois que le modèle améliore sa perte de validation :

```
1 | if val_loss < best_loss:  
2 |     generate_series(decoder) # Affiche une série de prénoms
```

### Affichage régulier des progrès :

- Le code affiche les pertes d'entraînement et de validation toutes les 500 époques :

```
1 | if epoch % 500 == 0:  
2 |     print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte entraînement :  
    {train_loss:.4f}, Perte validation : {val_loss:.4f}")
```

### Génération finale des prénoms :

- Après l'entraînement, une série finale de prénoms est générée :

```
1 | print("\nGénération finale de prénoms :")  
2 | generate_series(decoder, start_letters="JKLMNOP")
```

```
1 | try:  
2 |     import unicodecode  
3 | except ModuleNotFoundError:  
4 |     !pip install unicodecode  
5 |     import unicodecode  
6 |  
7 | import requests  
8 | import torch  
9 | import torch.nn as nn
```

```
10 from torch.autograd import Variable
11 import time
12 import math
13 import string
14 import random
15 import os
16
17 # Vérification GPU
18 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
19 print(f"Appareil utilisé : {device}")
20
21 # Téléchargement des données
22 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
23 data_dir = "data"
24 os.makedirs(data_dir, exist_ok=True)
25 data_path = os.path.join(data_dir, "russian.txt")
26
27 if not os.path.exists(data_path):
28     print("Téléchargement des données...")
29     response = requests.get(url)
30     with open(data_path, 'w', encoding='utf-8') as f:
31         f.write(response.text)
32
33 # Chargement des données
34 def unicode_to_ascii(s):
35     return ''.join(
36         c for c in unidecode.unidecode(s)
37         if c in (string.ascii_letters + " .,:;-")
38     )
39
40 def read_lines(filename):
41     with open(filename, encoding='utf-8') as f:
42         return [unicode_to_ascii(line.strip().lower()) for line in f]
43
44 lines = read_lines(data_path)
45 print(f"Nombre de prénoms : {len(lines)}")
46
47 # Division des données
48 random.shuffle(lines)
49 train_split = int(0.8 * len(lines))
50 valid_split = int(0.1 * len(lines))
51 train_lines = lines[:train_split]
52 valid_lines = lines[train_split:train_split + valid_split]
53 test_lines = lines[train_split + valid_split:]
54 print(f"Ensemble d'entraînement : {len(train_lines)}, validation : {len(valid_lines)}, Test : {len(test_lines)}")
55
56 # Paramètres globaux
57 all_letters = string.ascii_letters + " .,:;-"
```

```

58 n_letters = len(all_letters) + 1 # EOS marker
59 hidden_size = 128
60 n_layers = 2
61 lr = 0.005
62 bidirectional = True
63 max_length = 20
64
65 # Fonctions utilitaires
66 def char_tensor(string):
67     tensor = torch.zeros(len(string)).long()
68     for c in range(len(string)):
69         tensor[c] = all_letters.index(string[c])
70     return tensor
71
72 def input_tensor(line):
73     tensor = torch.zeros(len(line), 1, n_letters)
74     for li in range(len(line)):
75         letter = line[li]
76         tensor[li][0][all_letters.find(letter)] = 1
77     return tensor
78
79 def target_tensor(line):
80     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
81     letter_indexes.append(n_letters - 1) # EOS
82     return torch.LongTensor(letter_indexes)
83
84 def random_training_example(lines):
85     line = random.choice(lines)
86     input_line_tensor = input_tensor(line)
87     target_line_tensor = target_tensor(line)
88     return input_line_tensor, target_line_tensor
89
90 # Fonction pour afficher le temps écoulé
91 def time_since(since):
92     now = time.time()
93     s = now - since
94     m = math.floor(s / 60)
95     s -= m * 60
96     return f"{m}m {s:.2f}s"
97
98 # Définition du modèle
99 class RNNLight(nn.Module):
100     def __init__(self, input_size, hidden_size, output_size):
101         super(RNNLight, self).__init__()
102         self.input_size = input_size
103         self.hidden_size = hidden_size
104         self.output_size = output_size
105         self.bidirectional = bidirectional
106         self.num_directions = 2 if self.bidirectional else 1

```

```

107         self.rnn = nn.RNN(
108             input_size=input_size, hidden_size=hidden_size,
109             num_layers=1, bidirectional=self.bidirectional, batch_first=True
110         )
111         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
112         self.dropout = nn.Dropout(0.1)
113         self.softmax = nn.LogSoftmax(dim=1)
114
115     def forward(self, input, hidden):
116         _, hidden = self.rnn(input.unsqueeze(0), hidden)
117         hidden_concat = hidden if not self.bidirectional else
torch.cat((hidden[0], hidden[1]), 1)
118         output = self.out(hidden_concat)
119         output = self.dropout(output)
120         return self.softmax(output), hidden
121
122     def init_hidden(self):
123         return torch.zeros(self.num_directions, 1, self.hidden_size,
device=device)
124
125     # Entraînement avec sauvegarde
126     def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
criterion):
127         target_line_tensor = target_line_tensor.to(device) # Déplacement vers le bon
dispositif
128         hidden = decoder.init_hidden().to(device) # Initialisation sur le bon
dispositif
129         decoder.zero_grad()
130         loss = 0
131         for i in range(input_line_tensor.size(0)):
132             input_tensor = input_line_tensor[i].to(device) # Déplacement explicite
133             target_tensor = target_line_tensor[i].unsqueeze(0).to(device) #
Déplacement explicite
134             output, hidden = decoder(input_tensor, hidden.detach()) # Utilisation de
detach
135             l = criterion(output, target_tensor)
136             loss += l
137             loss.backward()
138             decoder_optimizer.step()
139             return loss.item() / input_line_tensor.size(0)
140
141     def validation(input_line_tensor, target_line_tensor, decoder, criterion):
142         with torch.no_grad(): # Pas de calcul de gradients pendant la validation
143             target_line_tensor = target_line_tensor.to(device)
144             hidden = decoder.init_hidden().to(device)
145             loss = 0
146             for i in range(input_line_tensor.size(0)):
147                 input_tensor = input_line_tensor[i].to(device)
148                 target_tensor = target_line_tensor[i].unsqueeze(0).to(device)

```

```

149         output, hidden = decoder(input_tensor, hidden.detach())
150         l = criterion(output, target_tensor)
151         loss += l
152         return loss.item() / input_line_tensor.size(0)
153
154 def sample(decoder, start_letter='A'):
155     """Génère un prénom à partir d'une lettre de départ."""
156     with torch.no_grad():
157         hidden = decoder.init_hidden()
158         input = input_tensor(start_letter)
159         output_name = start_letter
160         for _ in range(max_length):
161             output, hidden = decoder(input[0].to(device), hidden.to(device))
162             topi = output.topk(1)[1][0][0]
163             if topi == n_letters - 1:
164                 break
165             else:
166                 letter = all_letters[topi]
167                 output_name += letter
168                 input = input_tensor(letter)
169         return output_name
170
171 def generate_series(decoder, start_letters="ABCDE"):
172     """Génère une série de prénoms à partir de lettres de départ."""
173     print("Prénoms générés :")
174     for letter in start_letters:
175         print(f"- {sample(decoder, letter)}")
176
177 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
178 criterion):
179     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
180     start = time.time()
181     best_loss = float("inf")
182     model_path = "best_model_generation_prenom.pth"
183
184     for epoch in range(1, n_epochs + 1):
185         # Entraînement
186         input_line_tensor, target_line_tensor =
187         random_training_example(train_lines)
188         train_loss = train(input_line_tensor, target_line_tensor, decoder,
189 decoder_optimizer, criterion)
190
191         # validation
192         input_line_tensor, target_line_tensor =
193         random_training_example(valid_lines)
194         val_loss = validation(input_line_tensor, target_line_tensor, decoder,
195 criterion)
196
197         # Sauvegarde du meilleur modèle

```

```

193         if val_loss < best_loss:
194             best_loss = val_loss
195             torch.save(decoder.state_dict(), model_path)
196             print(f"\nÉpoque {epoch} : La perte de validation a diminué à
{best_loss:.4f}. Modèle sauvegardé.")
197             generate_series(decoder) # Affiche une série de prénoms
198
199         if epoch % 500 == 0:
200             print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte
entraînement : {train_loss:.4f}, Perte validation : {val_loss:.4f}")
201
202     # Exécution principale
203     if __name__ == "__main__":
204         decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
205         decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
206         criterion = nn.CrossEntropyLoss()
207         n_epochs = 10000 # Augmentez le nombre d'époques ici
208
209         print("Démarrage de l'entraînement...")
210         training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
211
212         print("\nGénération finale de prénoms :")
213         generate_series(decoder, start_letters="JKLMNOP")
214

```

### Version améliorée N°4 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V4.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V4.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

## Optimisations apportées

### 1. Précision ajoutée :

- La précision est calculée en divisant le nombre de prédictions correctes par le nombre total de caractères dans une séquence.
- Elle est affichée pendant l'entraînement et la validation.

### 2. Augmentation du nombre d'époques :

- Le nombre d'époques est défini à 200,000 :

```
1 | n_epochs = 200000
```

### 3. Ajustements des hyperparamètres :

- **Taille cachée ( `hidden_size` )** : Augmentée à 256 pour une meilleure capacité d'apprentissage.
- **Nombre de couches cachées ( `n_layers` )** : Augmenté à 3 pour une meilleure représentation des données.
- **Taux d'apprentissage ( `lr` )** : Réduit à 0.003 pour une convergence plus stable.

#### 4. Affichage des progrès :

- Tous les 500 époques, les pertes et précisions d'entraînement et de validation sont affichées.
- Lorsque le modèle est sauvegardé, la précision de validation est également affichée.

#### 5. Génération de prénoms :

- Une série de prénoms est générée et affichée chaque fois que le modèle s'améliore.

```

1  try:
2      import unicodecode
3  except ModuleNotFoundError:
4      !pip install unicodecode
5      import unicodecode
6
7  import requests
8  import torch
9  import torch.nn as nn
10 from torch.autograd import Variable
11 import time
12 import math
13 import string
14 import random
15 import os
16
17 # Vérification GPU
18 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
19 print(f"Appareil utilisé : {device}")
20
21 # Téléchargement des données
22 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
23 data_dir = "data"
24 os.makedirs(data_dir, exist_ok=True)
25 data_path = os.path.join(data_dir, "russian.txt")
26
27 if not os.path.exists(data_path):
28     print("Téléchargement des données...")
29     response = requests.get(url)
30     with open(data_path, 'w', encoding='utf-8') as f:
31         f.write(response.text)
32
33 # Chargement des données
34 def unicode_to_ascii(s):
35     return ''.join(
36         c for c in unicodecode.unidecode(s)

```

```

37         if c in (string.ascii_letters + " .,;'-")
38     )
39
40 def read_lines(filename):
41     with open(filename, encoding='utf-8') as f:
42         return [unicode_to_ascii(line.strip().lower()) for line in f]
43
44 lines = read_lines(data_path)
45 print(f"Nombre de prénoms : {len(lines)}")
46
47 # Division des données
48 random.shuffle(lines)
49 train_split = int(0.8 * len(lines))
50 valid_split = int(0.1 * len(lines))
51 train_lines = lines[:train_split]
52 valid_lines = lines[train_split:train_split + valid_split]
53 test_lines = lines[train_split + valid_split:]
54 print(f"Ensemble d'entraînement : {len(train_lines)}, validation :
55 {len(valid_lines)}, Test : {len(test_lines)}")
56
57 # Paramètres globaux
58 all_letters = string.ascii_letters + " .,;'-"
59 n_letters = len(all_letters) + 1 # EOS marker
60 bidirectional = True
61 max_length = 20
62
63 # Optimisation des paramètres et hyperparamètres
64 hidden_size = 256 # Augmentation de la taille cachée pour un meilleur
65 apprentissage
66 n_layers = 3 # Augmentation du nombre de couches cachées
67 lr = 0.003 # Ajustement du taux d'apprentissage
68 n_epochs = 200000 # Nombre d'époques augmenté
69
70 # Fonctions utilitaires
71 def char_tensor(string):
72     tensor = torch.zeros(len(string)).long()
73     for c in range(len(string)):
74         tensor[c] = all_letters.index(string[c])
75     return tensor
76
77 def input_tensor(line):
78     tensor = torch.zeros(len(line), 1, n_letters)
79     for li in range(len(line)):
80         letter = line[li]
81         tensor[li][0][all_letters.find(letter)] = 1
82     return tensor
83

```



```

84 def target_tensor(line):
85     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
86     letter_indexes.append(n_letters - 1) # EOS
87     return torch.LongTensor(letter_indexes)
88
89 def random_training_example(lines):
90     line = random.choice(lines)
91     input_line_tensor = input_tensor(line)
92     target_line_tensor = target_tensor(line)
93     return input_line_tensor, target_line_tensor
94
95 # Fonction pour afficher le temps écoulé
96 def time_since(since):
97     now = time.time()
98     s = now - since
99     m = math.floor(s / 60)
100    s -= m * 60
101    return f"{m}m {s:.2f}s"
102
103 # Définition du modèle
104 class RNNLight(nn.Module):
105     def __init__(self, input_size, hidden_size, output_size):
106         super(RNNLight, self).__init__()
107         self.input_size = input_size
108         self.hidden_size = hidden_size
109         self.output_size = output_size
110         self.bidirectional = bidirectional
111         self.num_directions = 2 if self.bidirectional else 1
112         self.rnn = nn.RNN(
113             input_size=input_size, hidden_size=hidden_size,
114             num_layers=1, bidirectional=self.bidirectional, batch_first=True
115         )
116         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
117         self.dropout = nn.Dropout(0.1)
118         self.softmax = nn.LogSoftmax(dim=1)
119
120     def forward(self, input, hidden):
121         _, hidden = self.rnn(input.unsqueeze(0), hidden)
122         hidden_concat = hidden if not self.bidirectional else
torch.cat((hidden[0], hidden[1]), 1)
123         output = self.out(hidden_concat)
124         output = self.dropout(output)
125         return self.softmax(output), hidden
126
127     def init_hidden(self):
128         return torch.zeros(self.num_directions, 1, self.hidden_size,
device=device)
129
130 # Entraînement avec sauvegarde

```

```

131 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
132 criterion):
133     target_line_tensor = target_line_tensor.to(device)
134     hidden = decoder.init_hidden().to(device)
135     decoder.zero_grad()
136     loss = 0
137     correct = 0 # Pour calculer la précision
138     total = target_line_tensor.size(0)
139
140     for i in range(input_line_tensor.size(0)):
141         input_tensor = input_line_tensor[i].to(device)
142         target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
143         output, hidden = decoder(input_tensor, hidden.detach())
144         l = criterion(output, target_tensor)
145         loss += l
146
147         # Calcul de la précision
148         predicted = output.topk(1)[1][0][0]
149         correct += (predicted == target_tensor[0]).item()
150
151     loss.backward()
152     decoder_optimizer.step()
153
154     accuracy = correct / total
155     return loss.item() / input_line_tensor.size(0), accuracy
156
157 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
158     with torch.no_grad():
159         target_line_tensor = target_line_tensor.to(device)
160         hidden = decoder.init_hidden().to(device)
161         loss = 0
162         correct = 0 # Pour calculer la précision
163         total = target_line_tensor.size(0)
164
165         for i in range(input_line_tensor.size(0)):
166             input_tensor = input_line_tensor[i].to(device)
167             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
168             output, hidden = decoder(input_tensor, hidden.detach())
169             l = criterion(output, target_tensor)
170             loss += l
171
172             # Calcul de la précision
173             predicted = output.topk(1)[1][0][0]
174             correct += (predicted == target_tensor[0]).item()
175
176         accuracy = correct / total
177         return loss.item() / input_line_tensor.size(0), accuracy

```

```

178 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
179 criterion):
180     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
181     start = time.time()
182     best_loss = float("inf")
183     model_path = "best_model_generation_prenom.pth"
184
185     for epoch in range(1, n_epochs + 1):
186         # Entraînement
187         input_line_tensor, target_line_tensor =
188         random_training_example(train_lines)
189         train_loss, train_acc = train(input_line_tensor, target_line_tensor,
190 decoder, decoder_optimizer, criterion)
191
192         # Validation
193         input_line_tensor, target_line_tensor =
194         random_training_example(valid_lines)
195         val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
196 decoder, criterion)
197
198         # Sauvegarde du meilleur modèle
199         if val_loss < best_loss:
200             best_loss = val_loss
201             torch.save(decoder.state_dict(), model_path)
202             print(f"\nÉpoch {epoch} : La perte de validation a diminué à
203 {best_loss:.4f}. Modèle sauvegardé.")
204             print(f"Précision de validation : {val_acc:.4f}")
205             generate_series(decoder)
206
207         if epoch % 500 == 0 or epoch == 1:
208             print(f"{time_since(start)} Époch {epoch}/{n_epochs}, Perte
209 entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
210             print(f"Perte validation : {val_loss:.4f}, Précision validation :
211 {val_acc:.4f}")
212
213     # Exécution principale
214     if __name__ == "__main__":
215         decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
216         decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
217         criterion = nn.CrossEntropyLoss()
218
219         print("Démarrage de l'entraînement...")
220         training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
221 criterion)
222
223         print("\nGénération finale de prénoms :")
224         generate_series(decoder, start_letters="JKLMNOP")

```

## Version améliorée N°5 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V5.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V5.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

```
1  import requests
2  import torch
3  import torch.nn as nn
4  from torch.autograd import Variable
5  import time
6  import math
7  import string
8  import random
9  import os
10 import matplotlib.pyplot as plt
11
12 # Vérification GPU
13 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
14 print(f"Appareil utilisé : {device}")
15
16 # Téléchargement des données
17 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
18 data_dir = "data"
19 os.makedirs(data_dir, exist_ok=True)
20 data_path = os.path.join(data_dir, "russian.txt")
21
22 if not os.path.exists(data_path):
23     print("Téléchargement des données...")
24     response = requests.get(url)
25     with open(data_path, 'w', encoding='utf-8') as f:
26         f.write(response.text)
27
28 # Chargement des données
29 def unicode_to_ascii(s):
30     return ''.join(
31         c for c in s if c in (string.ascii_letters + " .,:'-")
32     )
33
34 def read_lines(filename):
35     with open(filename, encoding='utf-8') as f:
36         return [unicode_to_ascii(line.strip().lower()) for line in f]
37
38 lines = read_lines(data_path)
39 print(f"Nombre de prénoms : {len(lines)}")
40
41 # Division des données
42 random.shuffle(lines)
```

```

43 train_split = int(0.7 * len(lines))
44 valid_split = int(0.2 * len(lines))
45 train_lines = lines[:train_split]
46 valid_lines = lines[train_split:train_split + valid_split]
47 test_lines = lines[train_split + valid_split:]
48 print(f"Ensemble d'entraînement : {len(train_lines)}, Validation :
    {len(valid_lines)}, Test : {len(test_lines)}")
49
50 # Paramètres globaux
51 all_letters = string.ascii_letters + " .,:'-"
52 n_letters = len(all_letters) + 1 # EOS marker
53 hidden_size = 256
54 n_layers = 3
55 lr = 0.003
56 bidirectional = True
57 max_length = 20
58 n_epochs = 200000
59
60 # Fonctions utilitaires
61 def char_tensor(string):
62     tensor = torch.zeros(len(string)).long()
63     for c in range(len(string)):
64         tensor[c] = all_letters.index(string[c])
65     return tensor
66
67 def input_tensor(line):
68     tensor = torch.zeros(len(line), 1, n_letters)
69     for li in range(len(line)):
70         letter = line[li]
71         tensor[li][0][all_letters.find(letter)] = 1
72     return tensor
73
74 def target_tensor(line):
75     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
76     letter_indexes.append(n_letters - 1) # EOS
77     return torch.LongTensor(letter_indexes)
78
79 def random_training_example(lines):
80     line = random.choice(lines)
81     input_line_tensor = input_tensor(line)
82     target_line_tensor = target_tensor(line)
83     return input_line_tensor, target_line_tensor
84
85 # Fonction pour afficher le temps écoulé
86 def time_since(since):
87     now = time.time()
88     s = now - since
89     m = math.floor(s / 60)
90     s -= m * 60

```

```

91         return f"{m}m {s:.2f}s"
92
93     # Définition du modèle
94     class RNNLight(nn.Module):
95         def __init__(self, input_size, hidden_size, output_size):
96             super(RNNLight, self).__init__()
97             self.input_size = input_size
98             self.hidden_size = hidden_size
99             self.output_size = output_size
100             self.bidirectional = bidirectional
101             self.num_directions = 2 if self.bidirectional else 1
102             self.rnn = nn.RNN(
103                 input_size=input_size, hidden_size=hidden_size,
104                 num_layers=n_layers, bidirectional=self.bidirectional,
105                 batch_first=True
106             )
107             self.out = nn.Linear(self.num_directions * hidden_size, output_size)
108             self.dropout = nn.Dropout(0.1)
109             self.softmax = nn.LogSoftmax(dim=1)
110
111         def forward(self, input, hidden):
112             _, hidden = self.rnn(input.unsqueeze(0), hidden)
113             hidden_concat = hidden if not self.bidirectional else
114             torch.cat((hidden[0], hidden[1]), 1)
115             output = self.out(hidden_concat)
116             output = self.dropout(output)
117             return self.softmax(output), hidden
118
119         def init_hidden(self):
120             return torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
121                                 device=device)
122
123     # Fonction pour générer des prénoms
124     def generate_prenoms(decoder, start_letters="ABCDE"):
125         print("\nPrénoms générés :")
126         for letter in start_letters:
127             print(f"- {sample(decoder, letter)}")
128
129     def sample(decoder, start_letter="A"):
130         with torch.no_grad():
131             hidden = decoder.init_hidden()
132             input = input_tensor(start_letter)
133             output_name = start_letter
134             for _ in range(max_length):
135                 output, hidden = decoder(input[0].to(device), hidden.to(device))
136                 topi = output.topk(1)[1][0][0]
137                 if topi == n_letters - 1:
138                     break
139             else:

```

```

137         letter = all_letters[topi]
138         output_name += letter
139         input = input_tensor(letter)
140         return output_name
141
142     # Entraînement avec sauvegarde
143     def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
144               criterion):
145         target_line_tensor = target_line_tensor.to(device)
146         hidden = decoder.init_hidden().to(device)
147         decoder.zero_grad()
148         loss = 0
149         correct = 0 # Précision
150         total = target_line_tensor.size(0)
151
152         for i in range(input_line_tensor.size(0)):
153             input_tensor = input_line_tensor[i].to(device)
154             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
155             output, hidden = decoder(input_tensor, hidden.detach())
156             l = criterion(output, target_tensor)
157             loss += l
158
159             # Calcul de la précision
160             predicted = output.topk(1)[1][0][0]
161             correct += (predicted == target_tensor[0]).item()
162
163         loss.backward()
164         decoder_optimizer.step()
165
166         accuracy = correct / total
167         return loss.item() / input_line_tensor.size(0), accuracy
168
169     def validation(input_line_tensor, target_line_tensor, decoder, criterion):
170         with torch.no_grad():
171             target_line_tensor = target_line_tensor.to(device)
172             hidden = decoder.init_hidden().to(device)
173             loss = 0
174             correct = 0
175             total = target_line_tensor.size(0)
176
177             for i in range(input_line_tensor.size(0)):
178                 input_tensor = input_line_tensor[i].to(device)
179                 target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
180                 output, hidden = decoder(input_tensor, hidden.detach())
181                 l = criterion(output, target_tensor)
182                 loss += l
183
184                 # Calcul de la précision
185                 predicted = output.topk(1)[1][0][0]

```

```

185         correct += (predicted == target_tensor[0]).item()
186
187     accuracy = correct / total
188     return loss.item() / input_line_tensor.size(0), accuracy
189
190 # Ajustement dynamique du taux d'apprentissage
191 def adjust_learning_rate(optimizer, epoch, decay_rate=0.5, step=20000):
192     if epoch % step == 0 and epoch > 0:
193         for param_group in optimizer.param_groups:
194             param_group['lr'] *= decay_rate
195             print(f"Taux d'apprentissage ajusté à : {param_group['lr']}")
196
197 # Fonction principale d'entraînement
198 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
199 criterion):
200     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
201     start = time.time()
202     best_loss = float("inf")
203     model_path = "best_model_generation_prenom.pth"
204
205     for epoch in range(1, n_epochs + 1):
206         adjust_learning_rate(decoder_optimizer, epoch)
207
208         input_line_tensor, target_line_tensor =
209 random_training_example(train_lines)
210         train_loss, train_acc = train(input_line_tensor, target_line_tensor,
211 decoder, decoder_optimizer, criterion)
212
213         input_line_tensor, target_line_tensor =
214 random_training_example(valid_lines)
215         val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
216 decoder, criterion)
217
218         if val_loss < best_loss:
219             best_loss = val_loss
220             torch.save(decoder.state_dict(), model_path)
221             print(f"\nÉpoch {epoch} : La perte de validation a diminué à
222 {best_loss:.4f}. Modèle sauvegardé.")
223             print(f"Précision validation : {val_acc:.4f}")
224             generate_prenoms(decoder)
225
226         if epoch % 500 == 0 or epoch == 1:
227             print(f"{time_since(start)} Époch {epoch}/{n_epochs}, Perte
228 entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
229             print(f"Perte validation : {val_loss:.4f}, Précision validation :
230 {val_acc:.4f}")
231
232 # Exécution principale
233 if __name__ == "__main__":

```



```

226     decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
227     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr,
weight_decay=1e-5)
228     criterion = nn.CrossEntropyLoss()
229
230     print("Démarrage de l'entraînement...")
231     training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
232

```

### Version améliorée N°6 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V7.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V7.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

Le modèle ajuste dynamiquement son taux d'apprentissage.

Génération de prénoms après chaque amélioration du modèle.

Visualisation en direct des pertes et précisions.

Évaluation finale avec la perte moyenne et la précision.

```

1  import requests
2  import torch
3  import torch.nn as nn
4  from torch.autograd import Variable
5  import time
6  import math
7  import string
8  import random
9  import os
10 import matplotlib.pyplot as plt
11
12 # Vérification GPU
13 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
14 print(f"Appareil utilisé : {device}")
15
16 # Téléchargement des données
17 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
18 data_dir = "data"
19 os.makedirs(data_dir, exist_ok=True)
20 data_path = os.path.join(data_dir, "russian.txt")
21
22 if not os.path.exists(data_path):

```

```

23     print("Téléchargement des données...")
24     response = requests.get(url)
25     with open(data_path, 'w', encoding='utf-8') as f:
26         f.write(response.text)
27
28     # Chargement des données
29     def unicode_to_ascii(s):
30         return ''.join(
31             c for c in s if c in (string.ascii_letters + " .,;'-")
32         )
33
34     def read_lines(filename):
35         with open(filename, encoding='utf-8') as f:
36             return [unicode_to_ascii(line.strip().lower()) for line in f]
37
38     lines = read_lines(data_path)
39     print(f"Nombre de prénoms : {len(lines)}")
40
41     # Division des données
42     random.shuffle(lines)
43     train_split = int(0.7 * len(lines))
44     valid_split = int(0.2 * len(lines))
45     train_lines = lines[:train_split]
46     valid_lines = lines[train_split:train_split + valid_split]
47     test_lines = lines[train_split + valid_split:]
48     print(f"Ensemble d'entraînement : {len(train_lines)}, Validation : {len(valid_lines)}, Test : {len(test_lines)}")
49
50     # Paramètres globaux
51     all_letters = string.ascii_letters + " .,;'-"
52     n_letters = len(all_letters) + 1 # EOS marker
53     hidden_size = 256
54     n_layers = 3
55     lr = 0.003
56     bidirectional = True
57     max_length = 20
58     n_epochs = 1000
59
60     # Fonctions utilitaires
61     def char_tensor(string):
62         tensor = torch.zeros(len(string)).long()
63         for c in range(len(string)):
64             tensor[c] = all_letters.index(string[c])
65         return tensor
66
67     def input_tensor(line):
68         tensor = torch.zeros(len(line), 1, n_letters)
69         for li in range(len(line)):
70             letter = line[li]

```

```

71     tensor[li][0][all_letters.find(letter)] = 1
72     return tensor
73
74 def target_tensor(line):
75     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
76     letter_indexes.append(n_letters - 1) # EOS
77     return torch.LongTensor(letter_indexes)
78
79 def random_training_example(lines):
80     line = random.choice(lines)
81     input_line_tensor = input_tensor(line)
82     target_line_tensor = target_tensor(line)
83     return input_line_tensor, target_line_tensor
84
85 # Fonction pour afficher le temps écoulé
86 def time_since(since):
87     now = time.time()
88     s = now - since
89     m = math.floor(s / 60)
90     s -= m * 60
91     return f"{m}m {s:.2f}s"
92
93 # Définition du modèle
94 class RNNLight(nn.Module):
95     def __init__(self, input_size, hidden_size, output_size):
96         super(RNNLight, self).__init__()
97         self.input_size = input_size
98         self.hidden_size = hidden_size
99         self.output_size = output_size
100         self.bidirectional = bidirectional
101         self.num_directions = 2 if self.bidirectional else 1
102         self.rnn = nn.RNN(
103             input_size=input_size, hidden_size=hidden_size,
104             num_layers=n_layers, bidirectional=self.bidirectional,
105             batch_first=True
106         )
107         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
108         self.dropout = nn.Dropout(0.1)
109         self.softmax = nn.LogSoftmax(dim=1)
110
111     def forward(self, input, hidden):
112         _, hidden = self.rnn(input.unsqueeze(0), hidden)
113         hidden_concat = hidden if not self.bidirectional else
114         torch.cat((hidden[0], hidden[1]), 1)
115         output = self.out(hidden_concat)
116         output = self.dropout(output)
117         return self.softmax(output), hidden
118
119     def init_hidden(self):

```

```

118         return torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
119                               device=device)
120
121     # Fonction pour générer des prénoms
122     def generate_prenoms(decoder, start_letters="ABCDE"):
123         print("\nPrénoms générés :")
124         for letter in start_letters:
125             print(f"- {sample(decoder, letter)}")
126
127     def sample(decoder, start_letter="A"):
128         with torch.no_grad():
129             hidden = decoder.init_hidden()
130             input = input_tensor(start_letter)
131             output_name = start_letter
132             for _ in range(max_length):
133                 output, hidden = decoder(input[0].to(device), hidden.to(device))
134                 topi = output.topk(1)[1][0][0]
135                 if topi == n_letters - 1:
136                     break
137                 else:
138                     letter = all_letters[topi]
139                     output_name += letter
140                     input = input_tensor(letter)
141             return output_name
142
143     # Entraînement avec sauvegarde
144     def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
145               criterion):
146         target_line_tensor = target_line_tensor.to(device)
147         hidden = decoder.init_hidden().to(device)
148         decoder.zero_grad()
149         loss = 0
150         correct = 0 # Précision
151         total = target_line_tensor.size(0)
152
153         for i in range(input_line_tensor.size(0)):
154             input_tensor = input_line_tensor[i].to(device)
155             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
156             output, hidden = decoder(input_tensor, hidden.detach())
157             l = criterion(output, target_tensor)
158             loss += l
159
160             # Calcul de la précision
161             predicted = output.topk(1)[1][0][0]
162             correct += (predicted == target_tensor[0]).item()
163
164         loss.backward()
165         decoder_optimizer.step()

```

```

165     accuracy = correct / total
166     return loss.item() / input_line_tensor.size(0), accuracy
167
168 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
169     with torch.no_grad():
170         target_line_tensor = target_line_tensor.to(device)
171         hidden = decoder.init_hidden().to(device)
172         loss = 0
173         correct = 0
174         total = target_line_tensor.size(0)
175
176         for i in range(input_line_tensor.size(0)):
177             input_tensor = input_line_tensor[i].to(device)
178             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
179             output, hidden = decoder(input_tensor, hidden.detach())
180             l = criterion(output, target_tensor)
181             loss += l
182
183             # Calcul de la précision
184             predicted = output.topk(1)[1][0][0]
185             correct += (predicted == target_tensor[0]).item()
186
187         accuracy = correct / total
188         return loss.item() / input_line_tensor.size(0), accuracy
189
190 # Ajustement dynamique du taux d'apprentissage
191 def adjust_learning_rate(optimizer, epoch, decay_rate=0.5, step=20000):
192     if epoch % step == 0 and epoch > 0:
193         for param_group in optimizer.param_groups:
194             param_group['lr'] *= decay_rate
195             print(f"Taux d'apprentissage ajusté à : {param_group['lr']}")
196
197 # Suivi des pertes et précisions
198 train_losses, val_losses = [], []
199 train_accuracies, val_accuracies = [], []
200
201 # Fonction principale d'entraînement
202 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
203             criterion):
204     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
205     start = time.time()
206     best_loss = float("inf")
207     model_path = "best_model_generation_prenom.pth"
208
209     for epoch in range(1, n_epochs + 1):
210         adjust_learning_rate(decoder_optimizer, epoch)
211
212         input_line_tensor, target_line_tensor =
random_training_example(train_lines)

```

```

212     train_loss, train_acc = train(input_line_tensor, target_line_tensor,
213     decoder, decoder_optimizer, criterion)
214     input_line_tensor, target_line_tensor =
215     random_training_example(valid_lines)
216     val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
217     decoder, criterion)
218
219     train_losses.append(train_loss)
220     val_losses.append(val_loss)
221     train_accuracies.append(train_acc)
222     val_accuracies.append(val_acc)
223
224     if val_loss < best_loss:
225         best_loss = val_loss
226         torch.save(decoder.state_dict(), model_path)
227         print(f"\nÉpoque {epoch} : La perte de validation a diminué à
228         {best_loss:.4f}. Modèle sauvegardé.")
229         print(f"Précision validation : {val_acc:.4f}")
230         generate_prenoms(decoder)
231
232     if epoch % 500 == 0 or epoch == 1:
233         print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte
234         entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
235         print(f"Perte validation : {val_loss:.4f}, Précision validation :
236         {val_acc:.4f}")
237
238     # Afficher les graphiques interactifs
239     plt.figure(figsize=(10, 5))
240     plt.plot(train_losses, label='Perte Entraînement')
241     plt.plot(val_losses, label='Perte Validation')
242     plt.legend()
243     plt.xlabel('Époques')
244     plt.ylabel('Perte')
245     plt.show()
246
247     plt.figure(figsize=(10, 5))
248     plt.plot(train_accuracies, label='Précision Entraînement')
249     plt.plot(val_accuracies, label='Précision Validation')
250     plt.legend()
251     plt.xlabel('Époques')
252     plt.ylabel('Précision')
253     plt.show()
254
255 # Évaluation finale
256 def evaluate_model(test_lines, decoder, criterion):
257     print("\n-----\n| ÉVALUATION FINALE |\n-----\n")
258     total_loss = 0
259     total_correct = 0

```

```

255     total_samples = 0
256     decoder.eval()
257
258     with torch.no_grad():
259         for line in test_lines:
260             input_line_tensor = input_tensor(line)
261             target_line_tensor = target_tensor(line)
262             loss, acc = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
263             total_loss += loss
264             total_correct += acc * len(line)
265             total_samples += len(line)
266
267     avg_loss = total_loss / len(test_lines)
268     avg_accuracy = total_correct / total_samples
269     print(f"Perte moyenne sur l'ensemble de test : {avg_loss:.4f}")
270     print(f"Précision moyenne sur l'ensemble de test : {avg_accuracy:.4f}")
271
272 # Exécution principale
273 if __name__ == "__main__":
274     decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
275     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr,
weight_decay=1e-5)
276     criterion = nn.CrossEntropyLoss()
277
278     print("Démarrage de l'entraînement...")
279     training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
280
281     print("\nChargement du meilleur modèle...")
282     # Chargement sécurisé pour éviter tout code malveillant
283     state_dict = torch.load("best_model_generation_prenom.pth",
map_location=device, weights_only=True)
284     decoder.load_state_dict(state_dict)
285     evaluate_model(test_lines, decoder, criterion)

```

## Version améliorée N°7 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V8.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V8.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

On souhaite seulement qu'à l'évaluation finale, Il générere 20 prénoms

```

1 import requests
2 import torch
3 import torch.nn as nn

```

```

4 from torch.autograd import Variable
5 import time
6 import math
7 import string
8 import random
9 import os
10 import matplotlib.pyplot as plt
11
12 # Vérification GPU
13 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
14 print(f"Appareil utilisé : {device}")
15
16 # Téléchargement des données
17 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
18 data_dir = "data"
19 os.makedirs(data_dir, exist_ok=True)
20 data_path = os.path.join(data_dir, "russian.txt")
21
22 if not os.path.exists(data_path):
23     print("Téléchargement des données...")
24     response = requests.get(url)
25     with open(data_path, 'w', encoding='utf-8') as f:
26         f.write(response.text)
27
28 # Chargement des données
29 def unicode_to_ascii(s):
30     return ''.join(
31         c for c in s if c in (string.ascii_letters + " .,;'-")
32     )
33
34 def read_lines(filename):
35     with open(filename, encoding='utf-8') as f:
36         return [unicode_to_ascii(line.strip().lower()) for line in f]
37
38 lines = read_lines(data_path)
39 print(f"Nombre de prénoms : {len(lines)}")
40
41 # Division des données
42 random.shuffle(lines)
43 train_split = int(0.7 * len(lines))
44 valid_split = int(0.2 * len(lines))
45 train_lines = lines[:train_split]
46 valid_lines = lines[train_split:train_split + valid_split]
47 test_lines = lines[train_split + valid_split:]
48 print(f"Ensemble d'entraînement : {len(train_lines)}, validation : {len(valid_lines)}, Test : {len(test_lines)}")
49
50 # Paramètres globaux
51 all_letters = string.ascii_letters + " .,;'-"

```



```

52 n_letters = len(all_letters) + 1 # EOS marker
53 hidden_size = 256
54 n_layers = 3
55 lr = 0.003
56 bidirectional = True
57 max_length = 20
58 n_epochs = 1000
59
60 # Fonctions utilitaires
61 def char_tensor(string):
62     tensor = torch.zeros(len(string)).long()
63     for c in range(len(string)):
64         tensor[c] = all_letters.index(string[c])
65     return tensor
66
67 def input_tensor(line):
68     tensor = torch.zeros(len(line), 1, n_letters)
69     for li in range(len(line)):
70         letter = line[li]
71         tensor[li][0][all_letters.find(letter)] = 1
72     return tensor
73
74 def target_tensor(line):
75     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
76     letter_indexes.append(n_letters - 1) # EOS
77     return torch.LongTensor(letter_indexes)
78
79 def random_training_example(lines):
80     line = random.choice(lines)
81     input_line_tensor = input_tensor(line)
82     target_line_tensor = target_tensor(line)
83     return input_line_tensor, target_line_tensor
84
85 # Fonction pour afficher le temps écoulé
86 def time_since(since):
87     now = time.time()
88     s = now - since
89     m = math.floor(s / 60)
90     s -= m * 60
91     return f"{m}m {s:.2f}s"
92
93 # Définition du modèle
94 class RNNLight(nn.Module):
95     def __init__(self, input_size, hidden_size, output_size):
96         super(RNNLight, self).__init__()
97         self.input_size = input_size
98         self.hidden_size = hidden_size
99         self.output_size = output_size
100         self.bidirectional = bidirectional

```

```

101         self.num_directions = 2 if self.bidirectional else 1
102         self.rnn = nn.RNN(
103             input_size=input_size, hidden_size=hidden_size,
104             num_layers=n_layers, bidirectional=self.bidirectional,
batch_first=True
105         )
106         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
107         self.dropout = nn.Dropout(0.1)
108         self.softmax = nn.LogSoftmax(dim=1)
109
110     def forward(self, input, hidden):
111         _, hidden = self.rnn(input.unsqueeze(0), hidden)
112         hidden_concat = hidden if not self.bidirectional else
torch.cat((hidden[0], hidden[1]), 1)
113         output = self.out(hidden_concat)
114         output = self.dropout(output)
115         return self.softmax(output), hidden
116
117     def init_hidden(self):
118         return torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
device=device)
119
120 # Fonction pour générer des prénoms
121 def generate_prenoms(decoder, start_letters="ABCDE"):
122     print("\nPrénoms générés :")
123     for letter in start_letters:
124         print(f"- {sample(decoder, letter)}")
125
126 def sample(decoder, start_letter="A"):
127     with torch.no_grad():
128         hidden = decoder.init_hidden()
129         input = input_tensor(start_letter)
130         output_name = start_letter
131         for _ in range(max_length):
132             output, hidden = decoder(input[0].to(device), hidden.to(device))
133             topi = output.topk(1)[1][0][0]
134             if topi == n_letters - 1:
135                 break
136             else:
137                 letter = all_letters[topi]
138                 output_name += letter
139                 input = input_tensor(letter)
140         return output_name
141
142 # Entraînement avec sauvegarde
143 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
criterion):
144     target_line_tensor = target_line_tensor.to(device)
145     hidden = decoder.init_hidden().to(device)

```

```

146     decoder.zero_grad()
147     loss = 0
148     correct = 0 # Précision
149     total = target_line_tensor.size(0)
150
151     for i in range(input_line_tensor.size(0)):
152         input_tensor = input_line_tensor[i].to(device)
153         target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
154         output, hidden = decoder(input_tensor, hidden.detach())
155         l = criterion(output, target_tensor)
156         loss += l
157
158         # Calcul de la précision
159         predicted = output.topk(1)[1][0][0]
160         correct += (predicted == target_tensor[0]).item()
161
162     loss.backward()
163     decoder_optimizer.step()
164
165     accuracy = correct / total
166     return loss.item() / input_line_tensor.size(0), accuracy
167
168 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
169     with torch.no_grad():
170         target_line_tensor = target_line_tensor.to(device)
171         hidden = decoder.init_hidden().to(device)
172         loss = 0
173         correct = 0
174         total = target_line_tensor.size(0)
175
176         for i in range(input_line_tensor.size(0)):
177             input_tensor = input_line_tensor[i].to(device)
178             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
179             output, hidden = decoder(input_tensor, hidden.detach())
180             l = criterion(output, target_tensor)
181             loss += l
182
183             # calcul de la précision
184             predicted = output.topk(1)[1][0][0]
185             correct += (predicted == target_tensor[0]).item()
186
187         accuracy = correct / total
188         return loss.item() / input_line_tensor.size(0), accuracy
189
190 # Ajustement dynamique du taux d'apprentissage
191 def adjust_learning_rate(optimizer, epoch, decay_rate=0.5, step=20000):
192     if epoch % step == 0 and epoch > 0:
193         for param_group in optimizer.param_groups:
194             param_group['lr'] *= decay_rate

```

```

195         print(f"Taux d'apprentissage ajusté à : {param_group['lr']}")
196
197     # Suivi des pertes et précisions
198     train_losses, val_losses = [], []
199     train_accuracies, val_accuracies = [], []
200
201     # Fonction principale d'entraînement
202     def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
203                 criterion):
204         print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
205         start = time.time()
206         best_loss = float("inf")
207         model_path = "best_model_generation_prenom.pth"
208
209         for epoch in range(1, n_epochs + 1):
210             adjust_learning_rate(decoder_optimizer, epoch)
211
212             input_line_tensor, target_line_tensor =
213             random_training_example(train_lines)
214             train_loss, train_acc = train(input_line_tensor, target_line_tensor,
215             decoder, decoder_optimizer, criterion)
216
217             input_line_tensor, target_line_tensor =
218             random_training_example(valid_lines)
219             val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
220             decoder, criterion)
221
222             train_losses.append(train_loss)
223             val_losses.append(val_loss)
224             train_accuracies.append(train_acc)
225             val_accuracies.append(val_acc)
226
227             if val_loss < best_loss:
228                 best_loss = val_loss
229                 torch.save(decoder.state_dict(), model_path)
230                 print(f"\nÉpoch {epoch} : La perte de validation a diminué à
231                 {best_loss:.4f}. Modèle sauvegardé.")
232                 print(f"Précision validation : {val_acc:.4f}")
233                 generate_prenoms(decoder)
234
235             if epoch % 500 == 0 or epoch == 1:
236                 print(f"{time_since(start)} Époch {epoch}/{n_epochs}, Perte
237                 entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
238                 print(f"Perte validation : {val_loss:.4f}, Précision validation :
239                 {val_acc:.4f}")
240
241             # Afficher les graphiques interactifs
242             plt.figure(figsize=(10, 5))
243             plt.plot(train_losses, label='Perte Entraînement')

```

```

236         plt.plot(val_losses, label='Perte Validation')
237         plt.legend()
238         plt.xlabel('Époques')
239         plt.ylabel('Perte')
240         plt.show()
241
242         plt.figure(figsize=(10, 5))
243         plt.plot(train_accuracies, label='Précision Entraînement')
244         plt.plot(val_accuracies, label='Précision Validation')
245         plt.legend()
246         plt.xlabel('Époques')
247         plt.ylabel('Précision')
248         plt.show()
249
250     # Évaluation finale
251     def evaluate_model(test_lines, decoder, criterion):
252         print("\n-----\n|  ÉVALUATION FINALE  |\n-----\n")
253         total_loss = 0
254         total_correct = 0
255         total_samples = 0
256         decoder.eval()
257
258         with torch.no_grad():
259             for line in test_lines:
260                 input_line_tensor = input_tensor(line)
261                 target_line_tensor = target_tensor(line)
262                 loss, acc = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
263                 total_loss += loss
264                 total_correct += acc * len(line)
265                 total_samples += len(line)
266
267         avg_loss = total_loss / len(test_lines)
268         avg_accuracy = total_correct / total_samples
269         print(f"Perte moyenne sur l'ensemble de test : {avg_loss:.4f}")
270         print(f"Précision moyenne sur l'ensemble de test : {avg_accuracy:.4f}")
271
272         # Génération de 20 prénoms avec le meilleur modèle
273         print("\nPrénoms générés avec le meilleur modèle :")
274         for _ in range(20):
275             start_letter = random.choice(all_letters) # Démarrer avec une lettre
aléatoire
276             print(f"- {sample(decoder, start_letter)}")
277
278
279     # Exécution principale
280     if __name__ == "__main__":
281         decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)

```

```

282     decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr,
weight_decay=1e-5)
283     criterion = nn.CrossEntropyLoss()
284
285     print("Démarrage de l'entraînement...")
286     training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
287
288     print("\nChargement du meilleur modèle...")
289     # Chargement sécurisé pour éviter tout code malveillant
290     state_dict = torch.load("best_model_generation_prenom.pth",
map_location=device, weights_only=True)
291     decoder.load_state_dict(state_dict)
292     evaluate_model(test_lines, decoder, criterion)

```

### Version améliorée N°8 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V10.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V10.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

Nous avons améliorée de nouveau le code de façon à mieux générer des prénoms russes.  
 Cette version est la meilleur au terme de génération

```

1  import requests
2  import torch
3  import torch.nn as nn
4  from torch.autograd import variable
5  import time
6  import math
7  import string
8  import random
9  import os
10 import matplotlib.pyplot as plt
11 import subprocess
12
13 # Vérification GPU
14 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
15 print(f"Appareil utilisé : {device}")
16
17 # Téléchargement des données
18 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
19 data_dir = "data"
20 os.makedirs(data_dir, exist_ok=True)
21 data_path = os.path.join(data_dir, "russian.txt")
22 shuffled_data_path = os.path.join(data_dir, "russian_shuffled.txt")
23
24 if not os.path.exists(data_path):

```

```

25     print("Téléchargement des données...")
26     response = requests.get(url)
27     with open(data_path, 'w', encoding='utf-8') as f:
28         f.write(response.text)
29
30
31
32
33
34 def shuffle_file(input_path, output_path):
35     """
36     Désordonne les lignes d'un fichier en utilisant la commande Bash `shuf`.
37     """
38     try:
39         subprocess.run(['shuf', input_path, '-o', output_path], check=True)
40         print(f"Fichier mélangé avec succès : {output_path}")
41     except FileNotFoundError:
42         print("Erreur : La commande `shuf` n'est pas disponible. Assurez-vous
43         qu'elle est installée.")
44         exit(1)
45
46
47
48
49
50
51
52 # Chargement des données
53 def unicode_to_ascii(s):
54     return ''.join(
55         c for c in s if c in (string.ascii_letters + " .,;'-")
56     )
57
58 def read_lines(filename):
59     with open(filename, encoding='utf-8') as f:
60         lines = f.readlines()
61
62     # Filtrer et nettoyer les lignes
63     clean_lines = []
64     for line in lines:
65         # Convertir en minuscules et supprimer les espaces autour
66         line = line.strip().lower()
67         # Vérifier que tous les caractères sont alphabétiques
68         if all(c in string.ascii_letters for c in line) and len(line) >= 3:
69             clean_lines.append(line)
70
71     # Supprimer les doublons et trier les prénoms
72     clean_lines = list(set(clean_lines))

```

```

73     clean_lines.sort()
74
75     return clean_lines
76
77
78     # Mélanger les lignes du fichier
79     shuffle_file(data_path, shuffled_data_path)
80
81     # Charger le fichier mélangé
82     lines = read_lines(shuffled_data_path)
83     print(f"Nombre de prénoms : {len(lines)}")
84
85     # Division des données
86     random.shuffle(lines)
87     train_split = int(0.7 * len(lines))
88     valid_split = int(0.2 * len(lines))
89     train_lines = lines[:train_split]
90     valid_lines = lines[train_split:train_split + valid_split]
91     test_lines = lines[train_split + valid_split:]
92     print(f"Ensemble d'entraînement : {len(train_lines)}, validation : {len(valid_lines)}, Test : {len(test_lines)}")
93
94     # Paramètres globaux
95     all_letters = string.ascii_letters + " .,:'-"
96     n_letters = len(all_letters) + 1 # EOS marker
97     hidden_size = 512
98     n_layers = 4
99     lr = 0.003
100    bidirectional = True
101    max_length = 20
102    n_epochs = 3000
103
104    # Fonctions utilitaires
105    def char_tensor(string):
106        tensor = torch.zeros(len(string)).long()
107        for c in range(len(string)):
108            tensor[c] = all_letters.index(string[c])
109        return tensor
110
111    def input_tensor(line):
112        tensor = torch.zeros(len(line), 1, n_letters)
113        for li in range(len(line)):
114            letter = line[li]
115            tensor[li][0][all_letters.find(letter)] = 1
116        return tensor
117
118    def target_tensor(line):
119        letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
120        letter_indexes.append(n_letters - 1) # EOS

```



```

121     return torch.LongTensor(letter_indexes)
122
123 def random_training_example(lines):
124     line = random.choice(lines)
125     input_line_tensor = input_tensor(line)
126     target_line_tensor = target_tensor(line)
127     return input_line_tensor, target_line_tensor
128
129 # Fonction pour afficher le temps écoulé
130 def time_since(since):
131     now = time.time()
132     s = now - since
133     m = math.floor(s / 60)
134     s -= m * 60
135     return f"{m}m {s:.2f}s"
136
137 # Définition du modèle
138 class RNNLight(nn.Module):
139     def __init__(self, input_size, hidden_size, output_size):
140         super(RNNLight, self).__init__()
141         self.input_size = input_size
142         self.hidden_size = hidden_size
143         self.output_size = output_size
144         self.bidirectional = bidirectional
145         self.num_directions = 2 if self.bidirectional else 1
146         self.rnn = nn.RNN(
147             input_size=input_size, hidden_size=hidden_size,
148             num_layers=n_layers, bidirectional=self.bidirectional,
149             batch_first=True
150         )
151         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
152         self.dropout = nn.Dropout(0.3)
153         self.softmax = nn.LogSoftmax(dim=1)
154
155     def forward(self, input, hidden):
156         _, hidden = self.rnn(input.unsqueeze(0), hidden)
157         hidden_concat = hidden if not self.bidirectional else
158         torch.cat((hidden[0], hidden[1]), 1)
159         output = self.out(hidden_concat)
160         output = self.dropout(output)
161         return self.softmax(output), hidden
162
163     def init_hidden(self):
164         return torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
165                             device=device)
166
167 # Fonction pour générer des prénoms
168 def generate_prenoms(decoder, start_letters="ABCDE"):
169     print("\nPrénoms générés :")

```

```

167     for letter in start_letters:
168         print(f"- {sample(decoder, letter)}")
169
170 def sample(decoder, start_letter="A", temperature=0.8):
171     with torch.no_grad():
172         hidden = decoder.init_hidden()
173         input = input_tensor(start_letter)
174         output_name = start_letter.lower() # Commencer en minuscule
175         for _ in range(max_length):
176             output, hidden = decoder(input[0].to(device), hidden.to(device))
177             # Appliquer la température
178             probabilities = torch.exp(output / temperature)
179             probabilities /= probabilities.sum() # Normaliser les probabilités
180             topi = torch.multinomial(probabilities, 1)[0][0] # Échantillonnage
181             if topi == n_letters - 1: # Fin de chaîne
182                 break
183             else:
184                 letter = all_letters[topi]
185                 if letter.isalpha(): # Garder uniquement les lettres
186                     output_name += letter.lower()
187                 else:
188                     break # Arrêter si un caractère non alphabétique est généré
189                 input = input_tensor(letter)
190         return output_name.capitalize()
191
192
193
194 # Entraînement avec sauvegarde
195 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
196 criterion):
197     target_line_tensor = target_line_tensor.to(device)
198     hidden = decoder.init_hidden().to(device)
199     decoder.zero_grad()
200     loss = 0
201     correct = 0 # Précision
202     total = target_line_tensor.size(0)
203
204     for i in range(input_line_tensor.size(0)):
205         input_tensor = input_line_tensor[i].to(device)
206         target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
207         output, hidden = decoder(input_tensor, hidden.detach())
208         l = criterion(output, target_tensor)
209         loss += l
210
211         # calcul de la précision
212         predicted = output.topk(1)[1][0][0]
213         correct += (predicted == target_tensor[0]).item()

```

```

214     loss.backward()
215     decoder_optimizer.step()
216
217     accuracy = correct / total
218     return loss.item() / input_line_tensor.size(0), accuracy
219
220 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
221     with torch.no_grad():
222         target_line_tensor = target_line_tensor.to(device)
223         hidden = decoder.init_hidden().to(device)
224         loss = 0
225         correct = 0
226         total = target_line_tensor.size(0)
227
228         for i in range(input_line_tensor.size(0)):
229             input_tensor = input_line_tensor[i].to(device)
230             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
231             output, hidden = decoder(input_tensor, hidden.detach())
232             l = criterion(output, target_tensor)
233             loss += l
234
235             # Calcul de la précision
236             predicted = output.topk(1)[1][0][0]
237             correct += (predicted == target_tensor[0]).item()
238
239         accuracy = correct / total
240         return loss.item() / input_line_tensor.size(0), accuracy
241
242 # Ajustement dynamique du taux d'apprentissage
243 def adjust_learning_rate(optimizer, epoch, decay_rate=0.5, step=20000):
244     if epoch % step == 0 and epoch > 0:
245         for param_group in optimizer.param_groups:
246             param_group['lr'] *= decay_rate
247             print(f"Taux d'apprentissage ajusté à : {param_group['lr']}")
248
249 # Suivi des pertes et précisions
250 train_losses, val_losses = [], []
251 train_accuracies, val_accuracies = [], []
252
253 # Fonction principale d'entraînement
254 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
255             criterion):
256     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
257     start = time.time()
258     best_loss = float("inf")
259     model_path = "best_model_generation_prenom.pth"
260
261     for epoch in range(1, n_epochs + 1):
262         adjust_learning_rate(decoder_optimizer, epoch)

```

```

262         input_line_tensor, target_line_tensor =
263         random_training_example(train_lines)
264         train_loss, train_acc = train(input_line_tensor, target_line_tensor,
265         decoder, decoder_optimizer, criterion)
266         input_line_tensor, target_line_tensor =
267         random_training_example(valid_lines)
268         val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
269         decoder, criterion)
270
271         train_losses.append(train_loss)
272         val_losses.append(val_loss)
273         train_accuracies.append(train_acc)
274         val_accuracies.append(val_acc)
275
276         if val_loss < best_loss:
277             best_loss = val_loss
278             torch.save(decoder.state_dict(), model_path)
279             print(f"\nÉpoque {epoch} : La perte de validation a diminué à
280             {best_loss:.4f}. Modèle sauvegardé.")
281             print(f"Précision validation : {val_acc:.4f}")
282             generate_prenoms(decoder)
283
284         if epoch % 500 == 0 or epoch == 1:
285             print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte
286             entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
287             print(f"Perte validation : {val_loss:.4f}, Précision validation :
288             {val_acc:.4f}")
289
290             # Afficher les graphiques interactifs
291             plt.figure(figsize=(10, 5))
292             plt.plot(train_losses, label='Perte Entraînement')
293             plt.plot(val_losses, label='Perte Validation')
294             plt.legend()
295             plt.xlabel('Époques')
296             plt.ylabel('Perte')
297             plt.show()
298
299             plt.figure(figsize=(10, 5))
300             plt.plot(train_accuracies, label='Précision Entraînement')
301             plt.plot(val_accuracies, label='Précision Validation')
302             plt.legend()
303             plt.xlabel('Époques')
304             plt.ylabel('Précision')
305             plt.show()
306
307     # Évaluation finale
308     def evaluate_model(test_lines, decoder, criterion):

```

```

304     print("\n-----\n|  ÉVALUATION FINALE | \n-----\n")
305     total_loss = 0
306     total_correct = 0
307     total_samples = 0
308     decoder.eval()
309
310     with torch.no_grad():
311         for line in test_lines:
312             input_line_tensor = input_tensor(line)
313             target_line_tensor = target_tensor(line)
314             loss, acc = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
315             total_loss += loss
316             total_correct += acc * len(line)
317             total_samples += len(line)
318
319     avg_loss = total_loss / len(test_lines)
320     avg_accuracy = total_correct / total_samples
321     print(f"Perte moyenne sur l'ensemble de test : {avg_loss:.4f}")
322     print(f"Précision moyenne sur l'ensemble de test : {avg_accuracy:.4f}")
323
324     # Génération de 20 prénoms uniques avec le meilleur modèle
325     print("\nPrénoms générés avec le meilleur modèle :")
326     generated_names = set()
327     attempts = 0 # Limiter les tentatives pour éviter les boucles infinies
328     while len(generated_names) < 20 and attempts < 50:
329         start_letter = random.choice(string.ascii_uppercase) # Démarrer avec une
lettre majuscule
330         name = sample(decoder, start_letter)
331         if len(name) >= 3: # Assurer une taille minimale de 3 lettres
332             generated_names.add(name)
333         attempts += 1
334
335     # Afficher les prénoms générés
336     for name in sorted(generated_names): # Trier pour lisibilité
337         print(f"- {name}")
338
339
340
341
342     # Exécution principale
343     if __name__ == "__main__":
344         decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
345         #decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr,
weight_decay=1e-5)
346         decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr,
weight_decay=1e-5)
347
348         criterion = nn.CrossEntropyLoss()

```

```

349
350     print("Démarrage de l'entraînement...")
351     training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
352
353     print("\nChargement du meilleur modèle...")
354     # Chargement sécurisé pour éviter tout code malveillant
355     state_dict = torch.load("best_model_generation_prenom.pth",
map_location=device, weights_only=True)
356     decoder.load_state_dict(state_dict)
357     evaluate_model(test_lines, decoder, criterion)

```

hidden\_size = 512

n\_layers = 4

lr = 0.003

Dropout(0.3)

train\_split = int(0.7 \* len(lines))

valid\_split = int(0.2 \* len(lines))

### Version améliorée N°9 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V11.ipynb.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V11.ipynb.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

Nous avons ajouté un test de couverture

```

1  import requests
2  import torch
3  import torch.nn as nn
4  from torch.autograd import Variable
5  import time
6  import math
7  import string
8  import random
9  import os
10 import matplotlib.pyplot as plt
11 import subprocess
12
13 # Vérification GPU
14 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
15 print(f"Appareil utilisé : {device}")
16
17 # Téléchargement des données
18 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
19 data_dir = "data"
20 os.makedirs(data_dir, exist_ok=True)
21 data_path = os.path.join(data_dir, "russian.txt")

```

```

22 shuffled_data_path = os.path.join(data_dir, "russian_shuffled.txt")
23
24 if not os.path.exists(data_path):
25     print("Téléchargement des données...")
26     response = requests.get(url)
27     with open(data_path, 'w', encoding='utf-8') as f:
28         f.write(response.text)
29
30
31
32
33
34 def shuffle_file(input_path, output_path):
35     """
36     Désordonne les lignes d'un fichier en utilisant la commande Bash `shuf`.
37     """
38     try:
39         subprocess.run(['shuf', input_path, '-o', output_path], check=True)
40         print(f"Fichier mélangé avec succès : {output_path}")
41     except FileNotFoundError:
42         print("Erreur : La commande `shuf` n'est pas disponible. Assurez-vous
43         qu'elle est installée.")
44         exit(1)
45
46
47
48
49
50
51
52 # Chargement des données
53 def unicode_to_ascii(s):
54     return ''.join(
55         c for c in s if c in (string.ascii_letters + " .,:'-")
56     )
57
58 def read_lines(filename):
59     with open(filename, encoding='utf-8') as f:
60         lines = f.readlines()
61
62     # Filtrer et nettoyer les lignes
63     clean_lines = []
64     for line in lines:
65         # Convertir en minuscules et supprimer les espaces autour
66         line = line.strip().lower()
67         # Vérifier que tous les caractères sont alphabétiques
68         if all(c in string.ascii_letters for c in line) and len(line) >= 3:
69             clean_lines.append(line)

```

```

70
71     # Supprimer les doublons et trier les prénoms
72     clean_lines = list(set(clean_lines))
73     clean_lines.sort()
74
75     return clean_lines
76
77
78 # Mélanger les lignes du fichier
79 shuffle_file(data_path, shuffled_data_path)
80
81 # Charger le fichier mélangé
82 lines = read_lines(shuffled_data_path)
83 print(f"Nombre de prénoms : {len(lines)}")
84
85 # Division des données
86 random.shuffle(lines)
87 train_split = int(0.7 * len(lines))
88 valid_split = int(0.2 * len(lines))
89 train_lines = lines[:train_split]
90 valid_lines = lines[train_split:train_split + valid_split]
91 test_lines = lines[train_split + valid_split:]
92 print(f"Ensemble d'entraînement : {len(train_lines)}, validation : {len(valid_lines)}, Test : {len(test_lines)}")
93
94 # Paramètres globaux
95 all_letters = string.ascii_letters + " .,;'-"
96 n_letters = len(all_letters) + 1 # EOS marker
97 hidden_size = 256
98 n_layers = 3
99 lr = 0.003
100 bidirectional = True
101 max_length = 20
102 n_epochs = 3000
103
104 # Fonctions utilitaires
105 def char_tensor(string):
106     tensor = torch.zeros(len(string)).long()
107     for c in range(len(string)):
108         tensor[c] = all_letters.index(string[c])
109     return tensor
110
111 def input_tensor(line):
112     tensor = torch.zeros(len(line), 1, n_letters)
113     for li in range(len(line)):
114         letter = line[li]
115         tensor[li][0][all_letters.find(letter)] = 1
116     return tensor
117

```



```

118 def target_tensor(line):
119     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
120     letter_indexes.append(n_letters - 1) # EOS
121     return torch.LongTensor(letter_indexes)
122
123 def random_training_example(lines):
124     line = random.choice(lines)
125     input_line_tensor = input_tensor(line)
126     target_line_tensor = target_tensor(line)
127     return input_line_tensor, target_line_tensor
128
129 # Fonction pour afficher le temps écoulé
130 def time_since(since):
131     now = time.time()
132     s = now - since
133     m = math.floor(s / 60)
134     s -= m * 60
135     return f"{m}m {s:.2f}s"
136
137 # Définition du modèle
138 class RNNLight(nn.Module):
139     def __init__(self, input_size, hidden_size, output_size):
140         super(RNNLight, self).__init__()
141         self.input_size = input_size
142         self.hidden_size = hidden_size
143         self.output_size = output_size
144         self.bidirectional = bidirectional
145         self.num_directions = 2 if self.bidirectional else 1
146         self.rnn = nn.RNN(
147             input_size=input_size, hidden_size=hidden_size,
148             num_layers=n_layers, bidirectional=self.bidirectional,
149             batch_first=True
150         )
151         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
152         self.dropout = nn.Dropout(0.1)
153         self.softmax = nn.LogSoftmax(dim=1)
154
155     def forward(self, input, hidden):
156         _, hidden = self.rnn(input.unsqueeze(0), hidden)
157         hidden_concat = hidden if not self.bidirectional else
158         torch.cat((hidden[0], hidden[1]), 1)
159         output = self.out(hidden_concat)
160         output = self.dropout(output)
161         return self.softmax(output), hidden
162
163     def init_hidden(self):
164         return torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
165                             device=device)

```

```

164 # Fonction pour générer des prénoms
165 def generate_prenoms(decoder, start_letters="ABCDE"):
166     print("\nPrénoms générés :")
167     for letter in start_letters:
168         print(f"- {sample(decoder, letter)}")
169
170 def sample(decoder, start_letter="A", temperature=0.8):
171     with torch.no_grad():
172         hidden = decoder.init_hidden()
173         input = input_tensor(start_letter)
174         output_name = start_letter.lower() # Commencer en minuscule
175         for _ in range(max_length):
176             output, hidden = decoder(input[0].to(device), hidden.to(device))
177             # Appliquer la température
178             probabilities = torch.exp(output / temperature)
179             probabilities /= probabilities.sum() # Normaliser les probabilités
180             topi = torch.multinomial(probabilities, 1)[0][0] # Échantillonnage
181             multinomial
182             if topi == n_letters - 1: # Fin de chaîne
183                 break
184             else:
185                 letter = all_letters[topi]
186                 if letter.isalpha(): # Garder uniquement les lettres
187                     output_name += letter.lower()
188                 else:
189                     break # Arrêter si un caractère non alphabétique est généré
190                 input = input_tensor(letter)
191             return output_name.capitalize()
192
193
194 # Entraînement avec sauvegarde
195 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
196 criterion):
197     target_line_tensor = target_line_tensor.to(device)
198     hidden = decoder.init_hidden().to(device)
199     decoder.zero_grad()
200     loss = 0
201     correct = 0 # Précision
202     total = target_line_tensor.size(0)
203
204     for i in range(input_line_tensor.size(0)):
205         input_tensor = input_line_tensor[i].to(device)
206         target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
207         output, hidden = decoder(input_tensor, hidden.detach())
208         l = criterion(output, target_tensor)
209         loss += l
210
211     # Calcul de la précision

```

```

211         predicted = output.topk(1)[1][0][0]
212         correct += (predicted == target_tensor[0]).item()
213
214     loss.backward()
215     decoder_optimizer.step()
216
217     accuracy = correct / total
218     return loss.item() / input_line_tensor.size(0), accuracy
219
220 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
221     with torch.no_grad():
222         target_line_tensor = target_line_tensor.to(device)
223         hidden = decoder.init_hidden().to(device)
224         loss = 0
225         correct = 0
226         total = target_line_tensor.size(0)
227
228         for i in range(input_line_tensor.size(0)):
229             input_tensor = input_line_tensor[i].to(device)
230             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
231             output, hidden = decoder(input_tensor, hidden.detach())
232             l = criterion(output, target_tensor)
233             loss += l
234
235             # Calcul de la précision
236             predicted = output.topk(1)[1][0][0]
237             correct += (predicted == target_tensor[0]).item()
238
239         accuracy = correct / total
240         return loss.item() / input_line_tensor.size(0), accuracy
241
242 # Ajustement dynamique du taux d'apprentissage
243 def adjust_learning_rate(optimizer, epoch, decay_rate=0.5, step=20000):
244     if epoch % step == 0 and epoch > 0:
245         for param_group in optimizer.param_groups:
246             param_group['lr'] *= decay_rate
247             print(f"Taux d'apprentissage ajusté à : {param_group['lr']}")
248
249 # Suivi des pertes et précisions
250 train_losses, val_losses = [], []
251 train_accuracies, val_accuracies = [], []
252
253 # Fonction principale d'entraînement
254 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
255             criterion):
256     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
257     start = time.time()
258     best_loss = float("inf")
259     model_path = "best_model_generation_prenom.pth"

```

```

259
260     for epoch in range(1, n_epochs + 1):
261         adjust_learning_rate(decoder_optimizer, epoch)
262
263         input_line_tensor, target_line_tensor =
random_training_example(train_lines)
264         train_loss, train_acc = train(input_line_tensor, target_line_tensor,
decoder, decoder_optimizer, criterion)
265
266         input_line_tensor, target_line_tensor =
random_training_example(valid_lines)
267         val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
decoder, criterion)
268
269         train_losses.append(train_loss)
270         val_losses.append(val_loss)
271         train_accuracies.append(train_acc)
272         val_accuracies.append(val_acc)
273
274         if val_loss < best_loss:
275             best_loss = val_loss
276             torch.save(decoder.state_dict(), model_path)
277             print(f"\nÉpoque {epoch} : La perte de validation a diminué à
{best_loss:.4f}. Modèle sauvegardé.")
278             print(f"Précision validation : {val_acc:.4f}")
279             generate_prenoms(decoder)
280
281         if epoch % 500 == 0 or epoch == 1:
282             print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte
entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
283             print(f"Perte validation : {val_loss:.4f}, Précision validation :
{val_acc:.4f}")
284
285         # Afficher les graphiques interactifs
286         plt.figure(figsize=(10, 5))
287         plt.plot(train_losses, label='Perte Entraînement')
288         plt.plot(val_losses, label='Perte Validation')
289         plt.legend()
290         plt.xlabel('Époques')
291         plt.ylabel('Perte')
292         plt.show()
293
294         plt.figure(figsize=(10, 5))
295         plt.plot(train_accuracies, label='Précision Entraînement')
296         plt.plot(val_accuracies, label='Précision Validation')
297         plt.legend()
298         plt.xlabel('Époques')
299         plt.ylabel('Précision')
300         plt.show()

```

```

301
302 # Évaluation finale
303 def evaluate_model(test_lines, decoder, criterion):
304     print("\n-----\n|  ÉVALUATION FINALE  |\n-----\n")
305     total_loss = 0
306     total_correct = 0
307     total_samples = 0
308     decoder.eval()
309
310     with torch.no_grad():
311         for line in test_lines:
312             input_line_tensor = input_tensor(line)
313             target_line_tensor = target_tensor(line)
314             loss, acc = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
315             total_loss += loss
316             total_correct += acc * len(line)
317             total_samples += len(line)
318
319     avg_loss = total_loss / len(test_lines)
320     avg_accuracy = total_correct / total_samples
321     print(f"Perte moyenne sur l'ensemble de test : {avg_loss:.4f}")
322     print(f"Précision moyenne sur l'ensemble de test : {avg_accuracy:.4f}")
323
324     # Génération de 20 prénoms uniques avec le meilleur modèle
325     print("\nPrénoms générés avec le meilleur modèle :")
326     generated_names = set()
327     attempts = 0 # Limiter les tentatives pour éviter les boucles infinies
328     while len(generated_names) < 20 and attempts < 50:
329         start_letter = random.choice(string.ascii_uppercase) # Démarrer avec une
lettre majuscule
330         name = sample(decoder, start_letter)
331         if len(name) >= 3: # Assurer une taille minimale de 3 lettres
332             generated_names.add(name)
333         attempts += 1
334
335     # Afficher les prénoms générés
336     for name in sorted(generated_names): # Trier pour lisibilité
337         print(f"- {name}")
338
339
340 # Test de couverture : Générer 10 000 prénoms et calculer le pourcentage dans le
corpus
341 def test_coverage(decoder, lines, num_samples=10000):
342     """
343     Génère `num_samples` prénoms et calcule le pourcentage de prénoms présents
dans le corpus.
344     """
345     print("\n-----\n|  TEST DE COUVERTURE  |\n-----\n")

```

```

346     generated_names = set()
347     corpus_set = set(lines) # Transformer les prénoms du corpus en un ensemble
pour une recherche rapide
348     matches = 0
349
350     for _ in range(num_samples):
351         start_letter = random.choice(string.ascii_uppercase) # Démarrer avec une
lettre majuscule
352         name = sample(decoder, start_letter)
353         if len(name) >= 3: # Vérifier que le prénom généré a au moins 3 lettres
354             generated_names.add(name)
355             if name.lower() in corpus_set: # Vérifier si le prénom est dans le
corpus (insensible à la casse)
356                 matches += 1
357
358         coverage = (matches / num_samples) * 100
359         print(f"Prénoms générés : {len(generated_names)} uniques sur {num_samples}
générés.")
360         print(f"Couverture : {coverage:.2f}% des prénoms générés sont présents dans le
corpus.")
361         return coverage
362
363
364
365 # Exécution principale
366 if __name__ == "__main__":
367     decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
368     #decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr,
weight_decay=1e-5)
369     decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr,
weight_decay=1e-5)
370
371     criterion = nn.CrossEntropyLoss()
372
373     print("Démarrage de l'entraînement...")
374     training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
375
376     print("\nChargement du meilleur modèle...")
377     # Chargement sécurisé pour éviter tout code malveillant
378     state_dict = torch.load("best_model_generation_prenom.pth",
map_location=device, weights_only=True)
379     decoder.load_state_dict(state_dict)
380     evaluate_model(test_lines, decoder, criterion)
381
382     # Appel du test de couverture
383     coverage = test_coverage(decoder, train_lines)

```

----- | TEST DE COUVERTURE | -----

Prénoms générés : 9295 uniques sur 10000 générés.

Couverture : 0.30% des prénoms générés sont présents dans le corpus.

```
1  import requests
2  import torch
3  import torch.nn as nn
4  from torch.autograd import Variable
5  import time
6  import math
7  import string
8  import random
9  import os
10 import matplotlib.pyplot as plt
11 import subprocess
12
13 # Vérification GPU
14 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
15 print(f"Appareil utilisé : {device}")
16
17 # Téléchargement des données
18 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
19 data_dir = "data"
20 os.makedirs(data_dir, exist_ok=True)
21 data_path = os.path.join(data_dir, "russian.txt")
22 shuffled_data_path = os.path.join(data_dir, "russian_shuffled.txt")
23
24 if not os.path.exists(data_path):
25     print("Téléchargement des données...")
26     response = requests.get(url)
27     with open(data_path, 'w', encoding='utf-8') as f:
28         f.write(response.text)
29
30
31
32
33
34 def shuffle_file(input_path, output_path):
35     """
36     Désordonne les lignes d'un fichier en utilisant la commande Bash `shuf`.
37     """
38     try:
39         subprocess.run(['shuf', input_path, '-o', output_path], check=True)
40         print(f"Fichier mélangé avec succès : {output_path}")
41     except FileNotFoundError:
```

```
42     print("Erreur : La commande `shuf` n'est pas disponible. Assurez-vous  
qu'elle est installée.")  
43     exit(1)  
44  
45  
46  
47  
48  
49  
50  
51  
52 # Chargement des données  
53 def unicode_to_ascii(s):  
54     return ''.join(  
55         c for c in s if c in (string.ascii_letters + " .,:'-")  
56     )  
57  
58 def read_lines(filename):  
59     with open(filename, encoding='utf-8') as f:  
60         lines = f.readlines()  
61  
62     # Filtrer et nettoyer les lignes  
63     clean_lines = []  
64     for line in lines:  
65         # Convertir en minuscules et supprimer les espaces autour  
66         line = line.strip().lower()  
67         # Vérifier que tous les caractères sont alphabétiques  
68         if all(c in string.ascii_letters for c in line) and len(line) >= 3:  
69             clean_lines.append(line)  
70  
71     # Supprimer les doublons et trier les prénoms  
72     clean_lines = list(set(clean_lines))  
73     clean_lines.sort()  
74  
75     return clean_lines  
76  
77  
78 # Mélanger les lignes du fichier  
79 shuffle_file(data_path, shuffled_data_path)  
80  
81 # Charger le fichier mélangé  
82 lines = read_lines(shuffled_data_path)  
83 print(f"Nombre de prénoms : {len(lines)}")  
84  
85 # Division des données  
86 random.shuffle(lines)  
87 train_split = int(0.8 * len(lines))  
88 valid_split = int(0.1 * len(lines))  
89 train_lines = lines[:train_split]
```



```

90 valid_lines = lines[train_split:train_split + valid_split]
91 test_lines = lines[train_split + valid_split:]
92 print(f"Ensemble d'entraînement : {len(train_lines)}, validation :
    {len(valid_lines)}, Test : {len(test_lines)}")
93
94 # Paramètres globaux
95 all_letters = string.ascii_letters + " .,:'-"
96 n_letters = len(all_letters) + 1 # EOS marker
97 hidden_size = 256
98 n_layers = 3
99 lr = 0.003
100 bidirectional = True
101 max_length = 20
102 n_epochs = 3000
103
104 # Fonctions utilitaires
105 def char_tensor(string):
106     tensor = torch.zeros(len(string)).long()
107     for c in range(len(string)):
108         tensor[c] = all_letters.index(string[c])
109     return tensor
110
111 def input_tensor(line):
112     tensor = torch.zeros(len(line), 1, n_letters)
113     for li in range(len(line)):
114         letter = line[li]
115         tensor[li][0][all_letters.find(letter)] = 1
116     return tensor
117
118 def target_tensor(line):
119     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
120     letter_indexes.append(n_letters - 1) # EOS
121     return torch.LongTensor(letter_indexes)
122
123 def random_training_example(lines):
124     line = random.choice(lines)
125     input_line_tensor = input_tensor(line)
126     target_line_tensor = target_tensor(line)
127     return input_line_tensor, target_line_tensor
128
129 # Fonction pour afficher le temps écoulé
130 def time_since(since):
131     now = time.time()
132     s = now - since
133     m = math.floor(s / 60)
134     s -= m * 60
135     return f"{m}m {s:.2f}s"
136
137 # Définition du modèle

```

```

138 class RNNLight(nn.Module):
139     def __init__(self, input_size, hidden_size, output_size):
140         super(RNNLight, self).__init__()
141         self.input_size = input_size
142         self.hidden_size = hidden_size
143         self.output_size = output_size
144         self.bidirectional = bidirectional
145         self.num_directions = 2 if self.bidirectional else 1
146         self.rnn = nn.RNN(
147             input_size=input_size, hidden_size=hidden_size,
148             num_layers=n_layers, bidirectional=self.bidirectional,
batch_first=True
149         )
150         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
151         self.dropout = nn.Dropout(0.1)
152         self.softmax = nn.LogSoftmax(dim=1)
153
154     def forward(self, input, hidden):
155         _, hidden = self.rnn(input.unsqueeze(0), hidden)
156         hidden_concat = hidden if not self.bidirectional else
torch.cat((hidden[0], hidden[1]), 1)
157         output = self.out(hidden_concat)
158         output = self.dropout(output)
159         return self.softmax(output), hidden
160
161     def init_hidden(self):
162         return torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
device=device)
163
164 # Fonction pour générer des prénoms
165 def generate_prenoms(decoder, start_letters="ABCDE"):
166     print("\nPrénoms générés :")
167     for letter in start_letters:
168         print(f"- {sample(decoder, letter)}")
169
170 def sample(decoder, start_letter="A", temperature=0.8):
171     with torch.no_grad():
172         hidden = decoder.init_hidden()
173         input = input_tensor(start_letter)
174         output_name = start_letter.lower() # Commencer en minuscule
175         for _ in range(max_length):
176             output, hidden = decoder(input[0].to(device), hidden.to(device))
177             # Appliquer la température
178             probabilities = torch.exp(output / temperature)
179             probabilities /= probabilities.sum() # Normaliser les probabilités
180             topi = torch.multinomial(probabilities, 1)[0][0] # Échantillonnage
multinomial
181             if topi == n_letters - 1: # Fin de chaîne
182                 break

```

```

183         else:
184             letter = all_letters[topi]
185             if letter.isalpha(): # Garder uniquement les lettres
186                 output_name += letter.lower()
187             else:
188                 break # Arrêter si un caractère non alphabétique est généré
189         input = input_tensor(letter)
190     return output_name.capitalize()
191
192
193
194 # Entraînement avec sauvegarde
195 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
196 criterion):
197     target_line_tensor = target_line_tensor.to(device)
198     hidden = decoder.init_hidden().to(device)
199     decoder.zero_grad()
200     loss = 0
201     correct = 0 # Précision
202     total = target_line_tensor.size(0)
203
204     for i in range(input_line_tensor.size(0)):
205         input_tensor = input_line_tensor[i].to(device)
206         target_tensor = target_line_tensor[i].unsqueeze(0).to(device)
207         output, hidden = decoder(input_tensor, hidden.detach())
208         l = criterion(output, target_tensor)
209         loss += l
210
211         # Calcul de la précision
212         predicted = output.topk(1)[1][0][0]
213         correct += (predicted == target_tensor[0]).item()
214
215     loss.backward()
216     decoder_optimizer.step()
217
218     accuracy = correct / total
219     return loss.item() / input_line_tensor.size(0), accuracy
220
221 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
222     with torch.no_grad():
223         target_line_tensor = target_line_tensor.to(device)
224         hidden = decoder.init_hidden().to(device)
225         loss = 0
226         correct = 0
227         total = target_line_tensor.size(0)
228
229         for i in range(input_line_tensor.size(0)):
230             input_tensor = input_line_tensor[i].to(device)
231             target_tensor = target_line_tensor[i].unsqueeze(0).to(device)

```

```

231         output, hidden = decoder(input_tensor, hidden.detach())
232         l = criterion(output, target_tensor)
233         loss += l
234
235         # Calcul de la précision
236         predicted = output.topk(1)[1][0][0]
237         correct += (predicted == target_tensor[0]).item()
238
239         accuracy = correct / total
240         return loss.item() / input_line_tensor.size(0), accuracy
241
242     # Ajustement dynamique du taux d'apprentissage
243     def adjust_learning_rate(optimizer, epoch, decay_rate=0.5, step=20000):
244         if epoch % step == 0 and epoch > 0:
245             for param_group in optimizer.param_groups:
246                 param_group['lr'] *= decay_rate
247                 print(f"Taux d'apprentissage ajusté à : {param_group['lr']}")
248
249     # Suivi des pertes et précisions
250     train_losses, val_losses = [], []
251     train_accuracies, val_accuracies = [], []
252
253     # Fonction principale d'entraînement
254     def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
255                 criterion):
256         print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
257         start = time.time()
258         best_loss = float("inf")
259         model_path = "best_model_generation_prenom.pth"
260
261         for epoch in range(1, n_epochs + 1):
262             adjust_learning_rate(decoder_optimizer, epoch)
263
264             input_line_tensor, target_line_tensor =
265             random_training_example(train_lines)
266             train_loss, train_acc = train(input_line_tensor, target_line_tensor,
267             decoder, decoder_optimizer, criterion)
268
269             input_line_tensor, target_line_tensor =
270             random_training_example(valid_lines)
271             val_loss, val_acc = validation(input_line_tensor, target_line_tensor,
272             decoder, criterion)
273
274             train_losses.append(train_loss)
275             val_losses.append(val_loss)
276             train_accuracies.append(train_acc)
277             val_accuracies.append(val_acc)
278
279             if val_loss < best_loss:

```

```

275         best_loss = val_loss
276         torch.save(decoder.state_dict(), model_path)
277         print(f"\nÉpoque {epoch} : La perte de validation a diminué à
{best_loss:.4f}. Modèle sauvegardé.")
278         print(f"Précision validation : {val_acc:.4f}")
279         generate_prenoms(decoder)
280
281     if epoch % 500 == 0 or epoch == 1:
282         print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte
entraînement : {train_loss:.4f}, Précision entraînement : {train_acc:.4f}")
283         print(f"Perte validation : {val_loss:.4f}, Précision validation :
{val_acc:.4f}")
284
285         # Afficher les graphiques interactifs
286         plt.figure(figsize=(10, 5))
287         plt.plot(train_losses, label='Perte Entraînement')
288         plt.plot(val_losses, label='Perte Validation')
289         plt.legend()
290         plt.xlabel('Époques')
291         plt.ylabel('Perte')
292         plt.show()
293
294         plt.figure(figsize=(10, 5))
295         plt.plot(train_accuracies, label='Précision Entraînement')
296         plt.plot(val_accuracies, label='Précision Validation')
297         plt.legend()
298         plt.xlabel('Époques')
299         plt.ylabel('Précision')
300         plt.show()
301
302     # Évaluation finale
303     def evaluate_model(test_lines, decoder, criterion):
304         print("\n-----\n|  ÉVALUATION FINALE  |\n-----\n")
305         total_loss = 0
306         total_correct = 0
307         total_samples = 0
308         decoder.eval()
309
310         with torch.no_grad():
311             for line in test_lines:
312                 input_line_tensor = input_tensor(line)
313                 target_line_tensor = target_tensor(line)
314                 loss, acc = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
315                 total_loss += loss
316                 total_correct += acc * len(line)
317                 total_samples += len(line)
318
319         avg_loss = total_loss / len(test_lines)

```

```

320     avg_accuracy = total_correct / total_samples
321     print(f"Perte moyenne sur l'ensemble de test : {avg_loss:.4f}")
322     print(f"Précision moyenne sur l'ensemble de test : {avg_accuracy:.4f}")
323
324     # Génération de 20 prénoms uniques avec le meilleur modèle
325     print("\nPrénoms générés avec le meilleur modèle :")
326     generated_names = set()
327     attempts = 0 # Limiter les tentatives pour éviter les boucles infinies
328     while len(generated_names) < 20 and attempts < 50:
329         start_letter = random.choice(string.ascii_uppercase) # Démarrer avec une
lettre majuscule
330         name = sample(decoder, start_letter)
331         if len(name) >= 3: # Assurer une taille minimale de 3 lettres
332             generated_names.add(name)
333             attempts += 1
334
335     # Afficher les prénoms générés
336     for name in sorted(generated_names): # Trier pour lisibilité
337         print(f"- {name}")
338
339
340 # Test de couverture : Générer 10 000 prénoms et calculer le pourcentage dans le
corpus
341 def test_coverage(decoder, lines, num_samples=10000):
342     """
343     Génère `num_samples` prénoms et calcule le pourcentage de prénoms présents
dans le corpus.
344     """
345     print("\n-----\n|  TEST DE COUVERTURE  |\n-----\n")
346     generated_names = set()
347     corpus_set = set(lines) # Transformer les prénoms du corpus en un ensemble
pour une recherche rapide
348     matches = 0
349
350     for _ in range(num_samples):
351         start_letter = random.choice(string.ascii_uppercase) # Démarrer avec une
lettre majuscule
352         name = sample(decoder, start_letter)
353         if len(name) >= 3: # Vérifier que le prénom généré a au moins 3 lettres
354             generated_names.add(name)
355             if name.lower() in corpus_set: # Vérifier si le prénom est dans le
corpus (insensible à la casse)
356                 matches += 1
357
358     coverage = (matches / num_samples) * 100
359     print(f"Prénoms générés : {len(generated_names)} uniques sur {num_samples}
générés.")
360     print(f"Couverture : {coverage:.2f}% des prénoms générés sont présents dans le
corpus.")

```

```

361     return coverage
362
363
364
365 # Exécution principale
366 if __name__ == "__main__":
367     decoder = RNNLight(n_letters, hidden_size, n_letters).to(device)
368     #decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr,
weight_decay=1e-5)
369     decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr,
weight_decay=1e-5)
370
371     criterion = nn.CrossEntropyLoss()
372
373     print("Démarrage de l'entraînement...")
374     training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
375
376     print("\nChargement du meilleur modèle...")
377     # Chargement sécurisé pour éviter tout code malveillant
378     state_dict = torch.load("best_model_generation_prenom.pth",
map_location=device, weights_only=True)
379     decoder.load_state_dict(state_dict)
380     evaluate_model(test_lines, decoder, criterion)
381
382     # Appel du test de couverture
383     coverage = test_coverage(decoder, train_lines)

```

### Version améliorée N°10 pour Colab Google

Dépôt: [https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation\\_prenoms\\_V11.ipynb.ipynb](https://github.com/olfabre/amsProjetMaster1/blob/olivier/Generation_prenoms_V11.ipynb.ipynb)

Data set: <https://olivier-fabre.com/passwordgenius/russian.txt>

nous l'avons amélioré avec un réseau de neurones LSTM

```

1  import requests
2  import torch
3  import torch.nn as nn
4  import time
5  import math
6  import string
7  import random
8  import os
9  import matplotlib.pyplot as plt
10 import subprocess
11 from torch.optim.lr_scheduler import ReduceLROnPlateau

```

```

12
13 # Vérification GPU
14 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
15 print(f"Appareil utilisé : {device}")
16
17 # Téléchargement des données
18 url = "https://olivier-fabre.com/passwordgenius/russian.txt"
19 data_dir = "data"
20 os.makedirs(data_dir, exist_ok=True)
21 data_path = os.path.join(data_dir, "russian.txt")
22 shuffled_data_path = os.path.join(data_dir, "russian_shuffled.txt")
23
24 if not os.path.exists(data_path):
25     print("Téléchargement des données...")
26     response = requests.get(url)
27     with open(data_path, 'w', encoding='utf-8') as f:
28         f.write(response.text)
29
30 def shuffle_file(input_path, output_path):
31     """
32     Désordonne les lignes d'un fichier en utilisant la commande Bash `shuf`.
33     """
34     try:
35         subprocess.run(['shuf', input_path, '-o', output_path], check=True)
36         print(f"Fichier mélangé avec succès : {output_path}")
37     except FileNotFoundError:
38         print("Erreur : La commande `shuf` n'est pas disponible. Assurez-vous
39         qu'elle est installée.")
40         exit(1)
41
42 # Chargement des données
43 def read_lines(filename):
44     with open(filename, encoding='utf-8') as f:
45         lines = f.readlines()
46
47     clean_lines = []
48     for line in lines:
49         line = line.strip().lower()
50         if all(c in string.ascii_letters for c in line) and len(line) >= 3:
51             clean_lines.append(line)
52
53     clean_lines = list(set(clean_lines))
54     clean_lines.sort()
55     return clean_lines
56
57 # Mélanger les lignes du fichier
58 shuffle_file(data_path, shuffled_data_path)
59
60 # Charger le fichier mélangé

```



```

60 lines = read_lines(shuffled_data_path)
61 print(f"Nombre de prénoms : {len(lines)}")
62
63 # Division des données
64 random.shuffle(lines)
65 train_split = int(0.8 * len(lines))
66 valid_split = int(0.1 * len(lines))
67 train_lines = lines[:train_split]
68 valid_lines = lines[train_split:train_split + valid_split]
69 test_lines = lines[train_split + valid_split:]
70 print(f"Ensemble d'entraînement : {len(train_lines)}, validation :
    {len(valid_lines)}, Test : {len(test_lines)}")
71
72 # Paramètres globaux
73 all_letters = string.ascii_letters + " .,:'-"
74 n_letters = len(all_letters) + 1
75 hidden_size = 256
76 n_layers = 3
77 lr = 0.003
78 bidirectional = True
79 max_length = 20
80 n_epochs = 15000
81
82 # Fonctions utilitaires
83 def char_tensor(string):
84     tensor = torch.zeros(len(string)).long()
85     for c in range(len(string)):
86         tensor[c] = all_letters.index(string[c])
87     return tensor
88
89 def input_tensor(line):
90     tensor = torch.zeros(len(line), 1, n_letters)
91     for li in range(len(line)):
92         letter = line[li]
93         tensor[li][0][all_letters.find(letter)] = 1
94     return tensor
95
96 def target_tensor(line):
97     letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
98     letter_indexes.append(n_letters - 1)
99     return torch.LongTensor(letter_indexes)
100
101 def random_training_example(lines):
102     line = random.choice(lines)
103     input_line_tensor = input_tensor(line)
104     target_line_tensor = target_tensor(line)
105     return input_line_tensor, target_line_tensor
106
107 def time_since(since):

```

```

108     now = time.time()
109     s = now - since
110     m = math.floor(s / 60)
111     s -= m * 60
112     return f"{m}m {s:.2f}s"
113
114 # Définition du modèle
115 class LSTMLight(nn.Module):
116     def __init__(self, input_size, hidden_size, output_size, n_layers=3,
117     bidirectional=True):
118         super(LSTMLight, self).__init__()
119         self.hidden_size = hidden_size
120         self.bidirectional = bidirectional
121         self.num_directions = 2 if self.bidirectional else 1
122         self.lstm = nn.LSTM(
123             input_size=input_size,
124             hidden_size=hidden_size,
125             num_layers=n_layers,
126             bidirectional=self.bidirectional,
127             batch_first=True,
128         )
129         self.out = nn.Linear(self.num_directions * hidden_size, output_size)
130         self.softmax = nn.LogSoftmax(dim=1)
131
132     def forward(self, input, hidden):
133         output, hidden = self.lstm(input.unsqueeze(0), hidden)
134         output = self.out(output.squeeze(0))
135         return self.softmax(output), hidden
136
137     def init_hidden(self):
138         return (
139             torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
140             device=device),
141             torch.zeros(self.num_directions * n_layers, 1, self.hidden_size,
142             device=device),
143         )
144
145 # Entraînement
146 def train(input_line_tensor, target_line_tensor, decoder, decoder_optimizer,
147 criterion):
148     hidden = decoder.init_hidden()
149     decoder_optimizer.zero_grad()
150     loss = 0
151
152     for i in range(input_line_tensor.size(0)):
153         output, hidden = decoder(input_line_tensor[i].to(device), hidden)
154         l = criterion(output, target_line_tensor[i].to(device).unsqueeze(0))
155         loss += l

```

```

153     loss.backward()
154     decoder_optimizer.step()
155     return loss.item() / input_line_tensor.size(0)
156
157 def validation(input_line_tensor, target_line_tensor, decoder, criterion):
158     with torch.no_grad():
159         hidden = decoder.init_hidden()
160         loss = 0
161
162         for i in range(input_line_tensor.size(0)):
163             output, hidden = decoder(input_line_tensor[i].to(device), hidden)
164             l = criterion(output, target_line_tensor[i].to(device).unsqueeze(0))
165             loss += l
166
167         return loss.item() / input_line_tensor.size(0)
168
169 # Fonction d'entraînement principale
170 def training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
171 criterion):
172     print("\n-----\n|  ENTRAÎNEMENT  |\n-----\n")
173     start = time.time()
174     best_loss = float("inf")
175     model_path = "best_model_generation_prenom.pth"
176     scheduler = ReduceLROnPlateau(decoder_optimizer, mode='min', factor=0.5,
177 patience=5, verbose=True)
178
179     train_losses, val_losses = [], []
180
181     for epoch in range(1, n_epochs + 1):
182         input_line_tensor, target_line_tensor =
183         random_training_example(train_lines)
184         train_loss = train(input_line_tensor, target_line_tensor, decoder,
185 decoder_optimizer, criterion)
186
187         input_line_tensor, target_line_tensor =
188         random_training_example(valid_lines)
189         val_loss = validation(input_line_tensor, target_line_tensor, decoder,
190 criterion)
191
192         scheduler.step(val_loss)
193
194         train_losses.append(train_loss)
195         val_losses.append(val_loss)
196
197         if val_loss < best_loss:
198             best_loss = val_loss
199             torch.save(decoder.state_dict(), model_path)
200             print(f"\nÉpoch {epoch} : La perte de validation a diminué à
201 {best_loss:.4f}. Modèle sauvegardé.")

```

```

195
196         if epoch % 500 == 0 or epoch == 1:
197             print(f"{time_since(start)} Époque {epoch}/{n_epochs}, Perte
entraînement : {train_loss:.4f}, Perte validation : {val_loss:.4f}")
198
199             # Affichage des graphiques
200             plt.figure(figsize=(10, 5))
201             plt.plot(train_losses, label="Perte d'entraînement")
202             plt.plot(val_losses, label="Perte de validation")
203             plt.legend()
204             plt.show()
205
206     # Génération de prénoms
207     def sample(decoder, start_letter="A", temperature=0.8):
208         with torch.no_grad():
209             hidden = decoder.init_hidden() # hidden est maintenant un tuple
(hidden_state, cell_state)
210             input = input_tensor(start_letter)
211             output_name = start_letter.lower() # Commencer en minuscule
212
213             for _ in range(max_length):
214                 output, hidden = decoder(input[0].to(device), (hidden[0].to(device),
hidden[1].to(device)))
215
216                 # Appliquer la température
217                 probabilities = torch.exp(output / temperature)
218                 probabilities /= probabilities.sum() # Normaliser les probabilités
219                 topi = torch.multinomial(probabilities, 1)[0][0] # Échantillonnage
multinomial
220
221                 if topi == n_letters - 1: # Fin de chaîne
222                     break
223                 else:
224                     letter = all_letters[topi]
225                     if letter.isalpha(): # Garder uniquement les lettres
226                         output_name += letter.lower()
227                     else:
228                         break # Arrêter si un caractère non alphabétique est généré
229
230                     input = input_tensor(letter)
231             return output_name.capitalize()
232
233     # Évaluation finale
234     def evaluate_model(test_lines, decoder, criterion):
235         print("\n-----\n| ÉVALUATION FINALE |\n-----\n")
236         total_loss = 0
237         total_correct = 0
238         total_samples = 0
239

```

```

240     with torch.no_grad():
241         for line in test_lines:
242             input_line_tensor = input_tensor(line)
243             target_line_tensor = target_tensor(line)
244             loss = validation(input_line_tensor, target_line_tensor, decoder,
criterion)
245             total_loss += loss
246             total_correct += 1 if loss < 0.5 else 0
247             total_samples += 1
248
249         avg_loss = total_loss / len(test_lines)
250         accuracy = total_correct / total_samples
251         print(f"Perte moyenne : {avg_loss:.4f}, Précision moyenne : {accuracy:.4f}")
252
253     # Test de couverture
254     def test_coverage(decoder, lines, num_samples=10000):
255         print("\n-----\n|  TEST DE COUVERTURE  |\n-----\n")
256         generated_names = set()
257         corpus_set = set(lines)
258         matches = 0
259
260         for _ in range(num_samples):
261             start_letter = random.choice(string.ascii_uppercase)
262             name = sample(decoder, start_letter)
263             if len(name) >= 3:
264                 generated_names.add(name)
265                 if name.lower() in corpus_set:
266                     matches += 1
267
268         coverage = (matches / num_samples) * 100
269         print(f"Prénoms générés : {len(generated_names)} uniques sur {num_samples}
générés.")
270         print(f"Couverture : {coverage:.2f}%")
271         return coverage
272
273     # Exécution principale
274     if __name__ == "__main__":
275         decoder = LSTMLight(n_letters, hidden_size, n_letters).to(device)
276         decoder_optimizer = torch.optim.AdamW(decoder.parameters(), lr=lr,
weight_decay=1e-5)
277         criterion = nn.CrossEntropyLoss()
278
279         print("Démarrage de l'entraînement...")
280         training(n_epochs, train_lines, valid_lines, decoder, decoder_optimizer,
criterion)
281
282         print("\nChargement du meilleur modèle...")
283         state_dict = torch.load("best_model_generation_prenom.pth",
map_location=device)

```

```
284     decoder.load_state_dict(state_dict)
285
286     evaluate_model(test_lines, decoder, criterion)
287     test_coverage(decoder, train_lines)
288
```

## Détail ligne par ligne du code

### 1-8 : Importations et vérification du GPU

- Importation des bibliothèques nécessaires : `requests` pour télécharger des données, `torch` pour PyTorch, `time` pour la gestion du temps, `math` pour les fonctions mathématiques, `string` pour manipuler des caractères, et `random` pour les opérations aléatoires.
- `torch.device` détermine si un GPU (CUDA) est disponible. Sinon, la CPU est utilisée. Cela optimise les calculs si un GPU est disponible.

### 9-22 : Téléchargement et configuration des fichiers

- Définit un chemin vers un fichier texte contenant des données.
- Vérifie si le fichier `russian.txt` existe déjà. Si non, télécharge son contenu depuis l'URL spécifiée et le sauvegarde localement.

### 23-32 : Fonction de mélange

- Implémente une fonction pour mélanger les lignes d'un fichier texte en utilisant la commande Bash `shuf`.
- Si `shuf` n'est pas installé, affiche un message d'erreur.

### 33-49 : Chargement et nettoyage des données

- `read_lines` charge un fichier ligne par ligne, nettoie chaque ligne (supprime les espaces inutiles, convertit en minuscules).
- Garde uniquement les lignes avec des caractères alphabétiques d'une certaine longueur ( $\geq 3$ ).
- Élimine les doublons et trie les lignes.

### 50-54 : Mélange et chargement des lignes

- Mélange les lignes du fichier source et charge les données mélangées en mémoire.

### 55-62 : Division des données

- Mélange les lignes aléatoirement.
- Divise les données en trois ensembles : entraînement (80%), validation (10%), et test (10%).

### 63-70 : Paramètres globaux

- Définit des variables pour les caractères acceptés, la taille des données d'entrée et de sortie, la structure du réseau (taille des couches cachées, directionnalité, etc.), et les paramètres d'entraînement (taux d'apprentissage, nombre d'époques).

### 71-93 : Fonctions utilitaires pour les tensors

- Transforme les chaînes de caractères en tenseurs utilisables par PyTorch pour les entrées et cibles du modèle.
- `random_training_example` sélectionne un exemple aléatoire pour l'entraînement.

#### 94-99 : Gestion du temps

- `time_since` mesure et formate la durée écoulée depuis un temps donné.

#### 100-120 : Définition du modèle LSTM

1 | LSTMlight

est une classe pour un réseau LSTM :

- Comporte une couche LSTM bidirectionnelle ou unidirectionnelle.
- Une couche linéaire transforme la sortie LSTM en une distribution sur les caractères possibles.
- Utilise une activation `LogSoftmax`.

#### 121-129 : Fonction d'entraînement

- Entraîne le modèle sur un exemple unique en calculant une perte, effectue une rétropropagation, et met à jour les paramètres.

#### 130-139 : Validation

- Évalue le modèle sur des données de validation sans rétropropagation pour mesurer la qualité du modèle.

#### 140-176 : Entraînement principal

- Gère plusieurs époques :
  - Entraîne sur les données d'entraînement.
  - Valide sur les données de validation.
  - Sauvegarde le modèle si la perte de validation s'améliore.
  - Réduit dynamiquement le taux d'apprentissage avec `ReduceLROnPlateau`.
  - Affiche les courbes de perte d'entraînement et de validation.

#### 177-200 : Génération de prénoms

- `sample` génère un prénom à partir d'une lettre initiale donnée en utilisant le modèle LSTM.
- Applique une température pour ajuster la créativité de la génération.

#### 201-217 : Évaluation finale

- Évalue la performance globale du modèle sur l'ensemble de test.
- Calcule la perte moyenne et une précision basée sur un seuil.

#### 218-232 : Test de couverture

- Génère des prénoms aléatoires, calcule combien apparaissent dans les données d'origine, et estime la "couverture".

### 233-243 : **Exécution principale**

- Crée une instance du modèle LSTM.
- Entraîne le modèle, charge le meilleur modèle sauvegardé, et l'évalue sur les ensembles de test.
- Effectue un test de couverture.

Ce code constitue une chaîne complète pour la génération de prénoms avec un LSTM, depuis la préparation des données jusqu'à l'évaluation finale.