
Modèles autorégressifs

Objectifs du chapitre

Dans ce chapitre, vous allez :

- Découvrez pourquoi les modèles autorégressifs sont bien adaptés à la génération de données séquentielles telles que comme texte.
- Apprenez à traiter et à tokeniser des données texte.
- Découvrez la conception architecturale des réseaux de neurones récurrents (RNN).
- Construisez et entraînez un réseau de mémoire à long terme (LSTM) à partir de zéro en utilisant Dur.
- Utilisez le LSTM pour générer un nouveau texte.
- Découvrez d'autres variantes de RNN, notamment les unités récurrentes fermées (GRU) et les cellules bidirectionnelles.
- Comprenez comment les données d'image peuvent être traitées comme une séquence de pixels.
- Découvrez la conception architecturale d'un PixelCNN.
- Créez un PixelCNN à partir de zéro en utilisant Keras.
- Utilisez PixelCNN pour générer des images.

Jusqu'à présent, nous avons exploré deux familles différentes de modèles génératifs qui impliquaient tous deux des variables latentes : les auto-encodeurs variationnels (VAE) et les réseaux contradictoires génératifs (GAN). Dans les deux cas, une nouvelle variable est introduite avec une distribution facile à échantillonner et le modèle apprend à décoder cette variable dans le domaine d'origine.

Nous allons maintenant nous intéresser aux modèles autorégressifs, une famille de modèles qui simplifient le problème de modélisation générative en le traitant comme un processus séquentiel. Les modèles autorégressifs conditionnent les prédictions sur les valeurs précédentes de la séquence,

plutôt que sur une variable aléatoire latente. Par conséquent, ils tentent de modéliser explicitement la distribution génératrice de données plutôt qu'une approximation de celle-ci (comme dans le cas des VAE).

Dans ce chapitre, nous explorerons deux modèles autorégressifs différents : les réseaux de mémoire à long terme et PixelCNN. Nous appliquerons le LSTM aux données texte et le PixelCNN aux données image. Nous aborderons en détail un autre modèle autorégressif très réussi, le Transformer, au [chapitre 9](#).

Introduction

Pour comprendre le fonctionnement d'un LSTM, nous allons d'abord visiter une étrange prison, où les détenus ont formé une société littéraire...

La Société littéraire des mécréants gênants

Edward Sopp détestait son travail de directeur de prison. Il passait ses journées à veiller sur les prisonniers et n'avait pas le temps de s'adonner à sa véritable passion : écrire des nouvelles. Il manquait d'inspiration et devait trouver un moyen de générer du nouveau contenu. Un jour, il eut une idée géniale qui lui permettrait de produire de nouvelles œuvres de fiction dans son style, tout en occupant les détenus : il demanderait aux détenus d'écrire collectivement les histoires pour lui ! Il a baptisé la nouvelle société la Société littéraire pour les mécréants gênants, ou LSTM ([Figure 5-1](#)).



Figure 5-1. Une grande cellule de prisonniers lisant des livres (créée avec [Midjourney](#))

La prison est particulièrement étrange car elle ne comprend qu'une seule grande cellule, contenant 256 prisonniers. Chaque prisonnier a une opinion sur la façon dont l'histoire actuelle d'Edward devrait continuer. Chaque jour, Edward publie le dernier mot de son roman dans la cellule, et il incombe aux détenus de mettre à jour individuellement leurs opinions sur l'état actuel de l'histoire, sur la base du nouveau mot et des opinions des détenus du précédent jour.

Réseau de mémoire à long terme (LSTM) |

Chaque prisonnier utilise un processus de pensée spécifique pour mettre à jour sa propre opinion, ce qui implique d'équilibrer les informations provenant du nouveau mot entrant et les opinions des autres prisonniers avec leurs propres croyances antérieures. Tout d'abord, ils décident quelle part de l'opinion d'hier ils souhaitent oublier, en tenant compte des informations contenues dans le nouveau mot et des opinions des autres prisonniers dans la cellule. Ils utilisent également ces informations pour former de nouvelles pensées et décident dans quelle mesure ils souhaitent les mélanger aux anciennes croyances qu'ils ont choisi de perpétuer de la veille. Cela forme alors la nouvelle opinion du prisonnier pour la journée.

Cependant, les prisonniers sont secrets et ne disent pas toujours toutes leurs opinions à leurs codétenus. Ils utilisent également chacun le dernier mot choisi et les opinions des autres détenus pour décider dans quelle mesure ils souhaitent divulguer leur opinion.

Lorsqu'Edward souhaite que la cellule génère le mot suivant de la séquence, les prisonniers communiquent chacun leurs opinions au gardien à la porte, qui combine ces informations pour finalement décider du mot suivant à ajouter à la fin du roman. Ce nouveau mot est ensuite réinjecté dans la cellule et le processus se poursuit jusqu'à ce que l'histoire complète soit terminée.

Pour former les détenus et le gardien, Edward alimente de courtes séquences de mots qu'il a précédemment écrits dans la cellule et vérifie si le mot suivant choisi par les détenus est correct. Il les informe de leur exactitude et, peu à peu, ils commencent à apprendre à écrire des histoires dans son propre style.

Après de nombreuses itérations de ce processus, Edward constate que le système est devenu assez performant pour générer un texte d'apparence réaliste. Satisfait des résultats, il publie un recueil des contes générés dans son nouveau livre, intitulé Les Fables d'E. Sopp.

L'histoire de M. Sopp et ses fables participatives est une analogie avec l'une des techniques autorégressives les plus connues pour les données séquentielles telles que le texte : le réseau de mémoire à long terme.

Réseau de mémoire à long terme (LSTM)

Un LSTM est un type particulier de réseau neuronal récurrent (RNN). Les RNN contiennent une couche (ou cellule) récurrente capable de gérer des données séquentielles en faisant en sorte que sa propre sortie à un pas de temps particulier fasse partie de l'entrée du pas de temps suivant.

Lorsque les RNN ont été introduits pour la première fois, les couches récurrentes étaient très simples et consistaient uniquement en un opérateur tanh qui garantissait que les informations transmises entre les pas de temps étaient mises à l'échelle entre -1 et 1 . Cependant, cette approche s'est avérée souffrir du problème du gradient de disparition et n'a pas fonctionné. Il ne s'adapte pas aux longues séquences de données.

Les cellules LSTM ont été introduites pour la première fois en 1997 dans un article de Sepp Hochreiter et Jürgen Schmidhuber.¹ Dans l'article, les auteurs décrivent comment les LSTM ne souffrent pas du problème.

même problème de gradient de disparition rencontré par les RNN vanille et peut être formé sur des séquences qui durent des centaines de pas de temps. Depuis lors, l'architecture LSTM a été adaptée et améliorée, et des variantes telles que les unités récurrentes fermées (abordées plus loin dans ce chapitre) sont désormais largement utilisées et disponibles sous forme de couches dans Keras.

Les LSTM ont été appliqués à un large éventail de problèmes impliquant des données séquentielles, notamment la prévision de séries chronologiques, l'analyse des sentiments et la classification audio. Dans ce chapitre, nous utiliserons les LSTM pour relever le défi de la génération de texte.



Exécution du code pour cet exemple

Le code de cet exemple se trouve dans le notebook Jupyter situé dans `notebooks/05_autoregressive/01_lstm/lstm.ipynb` dans le référentiel de livres.

L'ensemble de données de recettes

Nous utiliserons l'**ensemble de données Epicurious Recipes** qui est disponible via Kaggle. Il s'agit d'un ensemble de plus de 20 000 recettes, accompagnées de métadonnées telles que des informations nutritionnelles et des listes d'ingrédients.

Vous pouvez télécharger l'ensemble de données en exécutant le script de téléchargement de l'ensemble de données Kaggle dans le référentiel de livres, comme indiqué dans l'**exemple 5-1**. Cela enregistrera les recettes et les métadonnées qui les accompagnent localement dans le dossier `/data`.

Exemple 5-1. Téléchargement de l'ensemble de données Epicurious Recipe

```
bash scripts/download_kaggle_data.sh hugodarwood
```

épirecettes

L'**exemple 5-2** montre comment les données peuvent être chargées et filtrées afin de ne conserver que les recettes avec un titre et une description. Un exemple de chaîne de texte de recette est donné dans l'**exemple 5-3**.

Exemple 5-2. Chargement des données

```
avec open('/app/data/epirecipes/full_format_recipes.json')
comme json_data :      recette_data = json.load(json_data)
données_filtrées = [
    'Recette pour ' pour x      + x['titre']+ '      |      '      +      '.join(x['directions'])
    dans Recipe_data si 'titre' dans x et
    x['title'] n'est pas Aucun
```

et 'directions' en x et
x['directions'] n'est pas [Aucun](#)]

Exemple 5-3. Une chaîne de texte de l'ensemble de données Recettes

Recette de persillade de jambon avec salade de pommes de terre à la moutarde et purée de petits pois | Hachez suffisamment de feuilles de persil pour mesurer 1 cuillère à soupe ; réserve. Hacher les feuilles et les tiges restantes et laisser mijoter avec le bouillon et l'ail dans une petite casserole couverte pendant 5 minutes. Pendant ce temps, saupoudrez la gélatine sur l'eau dans un bol moyen et laissez ramollir 1 minute.

Passer le bouillon à travers un tamis à mailles fines dans un bol avec la gélatine et remuer pour dissoudre.

Assaisonnez avec du sel et du poivre. Placer le bol dans un bain de glace et laisser refroidir à température ambiante en remuant. Mélanger le jambon avec le persil réservé et répartir dans les bocaux. Versez la gélatine dessus et réfrigérez jusqu'à ce qu'elle soit prise, au moins 1 heure. Fouetter ensemble la mayonnaise, la moutarde, le vinaigre,

1/4 cuillère à café de sel et 1/4 cuillère à café de poivre dans un grand bol. Incorporer le céleri, les cornichons et les pommes de terre. Mélangez les pois avec la marjolaine, l'huile, 1/2 cuillère à café de poivre et 1/4 cuillère à café de sel dans un robot culinaire pour obtenir une purée grossière. Étalez les pois, puis la salade de pommes de terre sur le jambon.

Avant d'examiner comment construire un réseau LSTM dans Keras, nous devons d'abord faire un petit détour pour comprendre la structure des données texte et en quoi elles diffèrent des données d'image que nous avons vues jusqu'à présent dans ce livre.

Travailler avec des données texte

Il existe plusieurs différences clés entre les données texte et les données image, ce qui signifie que bon nombre des méthodes qui fonctionnent bien pour les données image ne sont pas aussi facilement applicables aux données texte. En particulier:

- Les données textuelles sont composées de blocs discrets (caractères ou mots), tandis que les pixels d'une image sont des points dans un spectre de couleurs continu. Nous pouvons facilement rendre un pixel vert plus bleu, mais il n'est pas évident de savoir comment procéder pour que le mot chat ressemble davantage au mot chien, par exemple. Cela signifie que nous pouvons facilement appliquer la rétropropagation aux données d'image, car nous pouvons calculer le gradient de notre fonction de perte par rapport aux pixels individuels pour établir la direction dans laquelle les couleurs des pixels doivent être modifiées pour minimiser la perte. Avec des données textuelles discrètes, nous ne pouvons évidemment pas appliquer la rétropropagation de la même manière, nous devons donc trouver un moyen de contourner ce problème.
- Les données texte ont une dimension temporelle mais pas de dimension spatiale, alors que les données d'image ont deux dimensions spatiales mais pas de dimension temporelle. L'ordre des mots est très important dans les données textuelles et les mots n'auraient aucun sens à l'envers, alors que les images peuvent généralement être inversées sans affecter le contenu. De plus, il existe souvent des dépendances séquentielles à long terme entre les mots qui doivent être capturées par le

modèle : par exemple, la réponse à une question ou la reprise du contexte d'un pronom. Avec les données d'image, tous les pixels peuvent être traités simultanément.

- Les données textuelles sont très sensibles aux petits changements dans les unités individuelles (mots ou caractères). Les données d'image sont généralement moins sensibles aux changements d'unités de pixels individuelles (une image d'une maison serait toujours reconnaissable comme une maison même si certains pixels étaient modifiés) mais avec les données de texte, changer même quelques mots peut considérablement modifier le sens du passage. , ou rendez-le absurde. Cela rend très difficile la formation d'un modèle pour générer un texte cohérent, car chaque mot est essentiel au sens global du passage.
- Les données texte ont une structure grammaticale basée sur des règles, alors que les données image ne suivent pas de règles définies sur la manière dont les valeurs de pixels doivent être attribuées. Par exemple, cela n'aurait aucun sens grammatical dans aucun contexte d'écrire « Le chat s'est assis sur l'avoir ». Il existe également des règles sémantiques extrêmement difficiles à modéliser ; cela n'aurait aucun sens de dire « Je suis à la plage », même si grammaticalement, il n'y a rien de mal à cette affirmation.



Progrès dans l'apprentissage profond génératif basé sur le texte

Jusqu'à récemment, la plupart des modèles d'apprentissage profond génératif les plus sophistiqués se concentraient sur les données d'images, car bon nombre des défis présentés dans la liste précédente étaient hors de portée, même des techniques les plus avancées. Cependant, au cours des cinq dernières années, des progrès étonnants ont été réalisés dans le domaine de l'apprentissage profond génératif basé sur du texte, grâce à l'introduction de l'architecture du modèle Transformer, que nous explorerons au [chapitre 9](#).

En gardant ces points à l'esprit, examinons maintenant les étapes à suivre pour mettre les données textuelles dans la bonne forme pour former un réseau LSTM.

Tokenisation

La première étape consiste à nettoyer et à tokeniser le texte. La tokenisation est le processus de division du texte en unités individuelles, telles que des mots ou des caractères.

La manière dont vous tokeniserez votre texte dépendra de ce que vous essayez d'atteindre avec votre modèle de génération de texte. Il y a des avantages et des inconvénients à utiliser à la fois des jetons de mots et de caractères, et votre choix affectera la manière dont vous devez nettoyer le texte avant la modélisation et la sortie de votre modèle.

Si vous utilisez des jetons de mots :

- Tout le texte peut être converti en minuscules, pour garantir que les mots en majuscules au début des phrases soient symbolisés de la même manière que les mêmes mots apparaissant au milieu d'une phrase. Dans certains cas, cependant, cela n'est pas souhaitable ; par exemple, certains noms propres, tels que des noms ou des lieux, peuvent bénéficier d'une majuscule afin d'être symbolisés indépendamment.
- Le vocabulaire du texte (l'ensemble des mots distincts dans l'ensemble d'apprentissage) peut être très vaste, certains mots apparaissant très peu ou peut-être une seule fois. Il peut être judicieux de remplacer les mots clairsemés par un jeton pour un mot inconnu, plutôt que de les inclure en tant que jetons distincts, afin de réduire le nombre de poids que le réseau neuronal doit apprendre.
- Les mots peuvent être radicalisés, ce qui signifie qu'ils sont réduits à leur forme la plus simple, de sorte que les différents temps d'un verbe restent symbolisés ensemble. Par exemple, parcourir, parcourir, parcourir et parcourir seraient tous liés aux sourcils.
- Vous devrez soit tokeniser la ponctuation, soit la supprimer complètement.
- L'utilisation de la tokenisation des mots signifie que le modèle ne pourra jamais prédire les mots en dehors du vocabulaire de formation.

Si vous utilisez des jetons de personnage :

- Le modèle peut générer des séquences de caractères qui forment de nouveaux mots en dehors du vocabulaire de formation – cela peut être souhaitable dans certains contextes, mais pas dans d'autres.
- Les lettres majuscules peuvent soit être converties en leurs homologues minuscules, soit rester sous forme de jetons séparés.
- Le vocabulaire est généralement beaucoup plus réduit lors de l'utilisation de la tokenisation des caractères. Ceci est bénéfique pour la vitesse de formation du modèle, car il y a moins de poids à apprendre dans la couche de sortie finale.

Pour cet exemple, nous utiliserons la tokenisation des mots minuscules, sans racine de mot.

Nous allons également tokeniser les signes de ponctuation, car nous aimerions que le modèle prédise quand il doit terminer les phrases ou utiliser des virgules, par exemple.

Le code de l'**exemple 5-4** nettoie et tokenise le texte. Exemple

5-4. Tokenisation

```
def pad_punctuation(s):                                s =
re.sub(r'([string.punctuation])', r'\1 ', s)

s = re.sub(' +', ' ', s)
Retour
```

```

text_data = [pad_punktion(x) pour x dans filtered_data] text_ds = ❶

tf.data.Dataset.from_tensor_slices(text_data).batch(32).shuffle(1000

) ❷

vectorize_layer = layer.TextVectorization( ❸
    standardize = 'inférieur', max_tokens
    = 10000, output_mode = "int",

    longueur_séquence_sortie =
    200 + 1, )

vectorize_layer.adapt(text_ds) ❹
vocabulaire =
vectorize_layer.get_vocabulary() ❺

```

- ❶ Complétez les signes de ponctuation pour les traiter comme des mots séparés.
- ❷ Convertir en un ensemble de données TensorFlow.
- ❸ Créez un calque Keras TextVectorization pour convertir le texte en minuscules, attribuez aux 10 000 mots les plus courants un jeton entier correspondant et coupez ou complétez la séquence à 201 jetons.
- ❹ Appliquez la couche TextVectorization aux données d'entraînement.
- ❺ La variable de vocabulaire stocke une liste de jetons de mots.

Un exemple de recette après tokenisation est présenté dans l'exemple 5-5. La longueur de la séquence que nous utilisons pour entraîner le modèle est un paramètre du processus de formation. Dans cet exemple, nous choisissons d'utiliser une longueur de séquence de 200, donc nous complétons ou découpons la recette à une longueur de plus que cette longueur, pour nous permettre de créer la variable cible (plus d'informations à ce sujet dans la section suivante). Pour obtenir la longueur souhaitée, la fin du vecteur est complétée par des zéros.



Jetons d'arrêt

Le jeton 0 est appelé jeton d'arrêt, ce qui signifie que la chaîne de texte est terminée.

Exemple 5-5. La recette de l'exemple 5-3 tokenisée

```
[ 26 16 557          1      8 298 335 189          4 1054 494 27 332 228
```

Réseau de mémoire à long terme (LSTM) |

235 262	5 594 11 133 22 311	2 332 45 262	4 671
4 70	8 171 4 81	9 65 80 3 121	3 59
12 2 299	3 88 650 20	39 6 9 29 21	4 67
529 11 164	2 320 171 102	9 374 13 643 306 25 21	
8 650 4 42	5 931 2	63 8 24 4 33	2 114
21 6 178 181 1245	4 60	5 140 112 3 48	2 117
557 8 285 235	4 200 292 980	2 107 650 28 72	4
108 10 114	3 57 204 11 172	2 73 110 482	3 298
3 190 33 3 11	23 32 142 24 42	4 11 23 32 142	
6 9 30 21 2	3 6 353 3 3224 4 150 3		
2 437 494 8 1281	3 37	3 11 23 15 142 33	3
4 11 23 32 142 24	6	9 291 188 5 9 412 572	
2 230 494 3 46 335 189		3 20 557 2 0 0 0	
0 0 0 0 0]			

Dans l'exemple 5-6, nous pouvons voir un sous-ensemble de la liste des jetons mappés à leurs indices respectifs. La couche réserve le jeton 0 pour le remplissage (c'est-à-dire le jeton d'arrêt) et le jeton 1 pour les mots inconnus qui ne font pas partie des 10 000 premiers mots (par exemple, persillade). Les autres mots se voient attribuer des jetons par ordre de fréquence. Le nombre de mots à inclure dans le vocabulaire est également un paramètre du processus de formation. Plus il y a de mots inclus, moins vous verrez de jetons inconnus dans le texte ; cependant, votre modèle devra être plus grand pour s'adapter à la plus grande taille de vocabulaire.

Exemple 5-6. Le vocabulaire de la couche TextVectorization

```
0 :
1 : [INCONNUE]
2 : .
3 : ,
4 : et
5 : à
6 : dans
7 : le
8 : avec
9 : un
```

Création de l'ensemble de formation

Notre LSTM sera entraîné à prédire le mot suivant dans une séquence, étant donné une séquence de mots précédant ce point. Par exemple, nous pourrions donner au modèle les jetons pour le poulet grillé avec bouilli et nous attendre à ce que le modèle produise un mot suivant approprié (par exemple, pommes de terre plutôt que bananes).

Nous pouvons donc simplement décaler toute la séquence d'un jeton afin de créer notre variable cible.

L'étape de génération de l'ensemble de données peut être réalisée avec le code de l'exemple 5-7. Exemple

5-7. Création de l'ensemble de données de formation

```
def prepare_inputs(texte):
    tf.expand_dims(texte, -1)
    tokenized_sentences =
    vectorize_layer(texte)
    x = phrases_tokenisées[:, :-1]
    y = tokenized_sentences[:, 1:]
    return x, y

train_ds = text_ds.map(prepare_inputs)
```

- 1 Créez l'ensemble d'entraînement composé de jetons de recette (l'entrée) et du même vecteur décalé d'un jeton (la cible).

L'architecture LSTM

L'architecture du modèle LSTM global est présentée dans le Tableau 5-1. L'entrée du modèle est une séquence de jetons entiers et la sortie est la probabilité que chaque mot du vocabulaire de 10 000 mots apparaisse ensuite dans la séquence. Pour comprendre comment cela fonctionne en détail, nous devons introduire deux nouveaux types de couches, Embedding et LSTM.

Tableau 5-1. Résumé du modèle du LSTM

Numéro de paramètre de sortie de couche		
(taper)	forme	
InputLayer	(Aucun, Aucun)	0
Incorporation	(Aucun, Aucun, 100)	1 000 000
LSTM	(Aucun, Aucun, 128)	117 248
Dense	(Aucun, Aucun, 10 000)	1 290 000
Total des paramètres 2 407 248		
Paramètres entraînaables 2 407 248		
Paramètres non entraînaables 0		



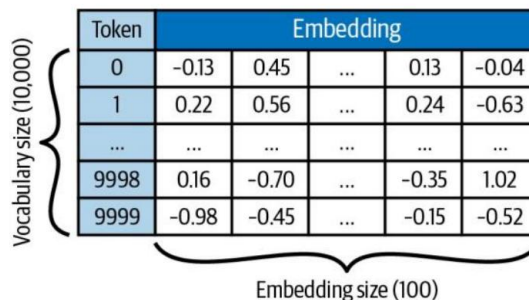
La couche d'entrée du LSTM

Notez que la couche d'entrée n'a pas besoin que nous spécifiions la longueur de la séquence à l'avance. La taille du lot et la longueur de la séquence sont flexibles (d'où la forme (Aucun, Aucun)). C'est parce que tout

les couches en aval sont indépendantes de la longueur de la séquence traversée.

La couche d'intégration

Une couche d'intégration est essentiellement une table de recherche qui convertit chaque jeton entier en un vecteur de longueur `embedding_size`, comme le montre la [figure 5-2](#). Les vecteurs de recherche sont appris par le modèle sous forme de poids. Par conséquent, le nombre de poids appris par cette couche est égal à la taille du vocabulaire multiplié par la dimension du vecteur d'intégration (c'est-à-dire $10\,000 \times 100 = 1\,000\,000$).



Token	Embedding				
0	-0.13	0.45	...	0.13	-0.04
1	0.22	0.56	...	0.24	-0.63
...
9998	0.16	-0.70	...	-0.35	1.02
9999	-0.98	-0.45	...	-0.15	-0.52

Figure 5-2. Une couche d'intégration est une table de recherche pour chaque jeton entier

Nous intégrons chaque jeton entier dans un vecteur continu car cela permet au modèle d'apprendre une représentation pour chaque mot qui peut être mise à jour par rétropropagation. Nous pourrions également encoder à chaud chaque jeton d'entrée, mais l'utilisation d'une couche d'intégration est préférable car elle rend l'intégration elle-même entraînable, donnant ainsi au modèle plus de flexibilité pour décider comment intégrer chaque jeton pour améliorer ses performances.

Par conséquent, la couche d'entrée transmet un tenseur de séquences entières de forme `[batch_size, seq_length]` à la couche Embedding, qui génère un tenseur de forme `[batch_size, seq_length, embedding_size]`. Ceci est ensuite transmis à la couche LSTM ([Figure 5-3](#)).

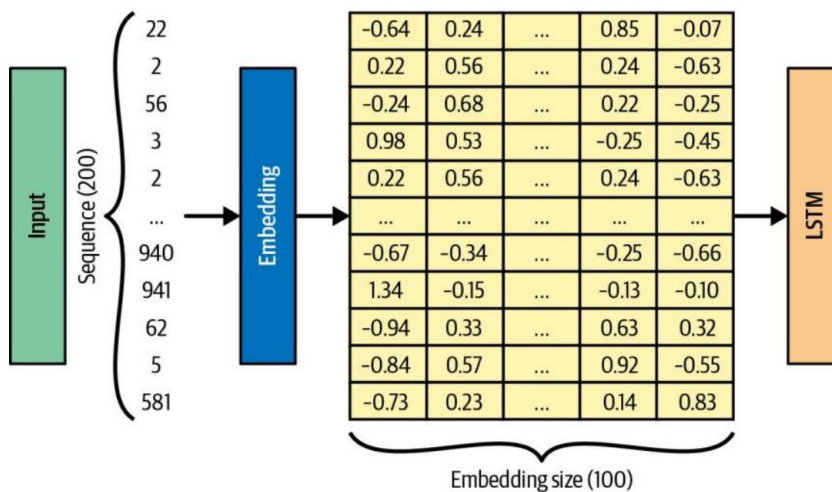


Figure 5-3. Une seule séquence lorsqu'elle traverse une couche d'intégration

La couche LSTM

Pour comprendre la couche LSTM, il faut d'abord regarder comment fonctionne une couche récurrente générale.

Une couche récurrente a la propriété particulière de pouvoir traiter des données d'entrée séquentielles x_1, \dots, x_n . Il s'agit d'une cellule qui met à jour son état caché, h_t , à mesure que chaque élément de la séquence x_t y passe, un pas de temps à la fois.

L'état caché est un vecteur dont la longueur est égale au nombre d'unités dans la cellule. Il peut être considéré comme la compréhension actuelle de la séquence par la cellule. Au pas de temps t , la cellule utilise la valeur précédente de l'état caché, h_{t-1} , ainsi que les données du pas de temps actuel x_t pour produire un vecteur d'état caché mis à jour, h_t . Ce processus récurrent se poursuit jusqu'à la fin de la séquence. Une fois la séquence terminée, la couche génère l'état caché final de la cellule, h_n , qui est ensuite transmis à la couche suivante du réseau. Ce processus est illustré à la figure 5-4.

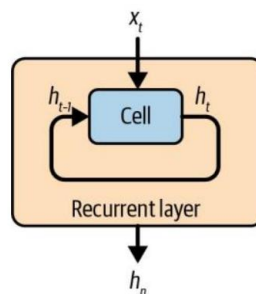


Figure 5-4. Un schéma simple d'une couche récurrente

Pour expliquer cela plus en détail, déroulons le processus afin que nous puissions voir exactement comment une seule séquence est transmise à travers la couche (Figure 5-5).



Poids des cellules

Il est important de se rappeler que toutes les cellules de ce diagramme partagent les mêmes poids (car il s'agit en réalité de la même cellule). Il n'y a aucune différence entre ce diagramme et la figure 5-4 ; c'est juste une manière différente de dessiner la mécanique d'une couche récurrente.

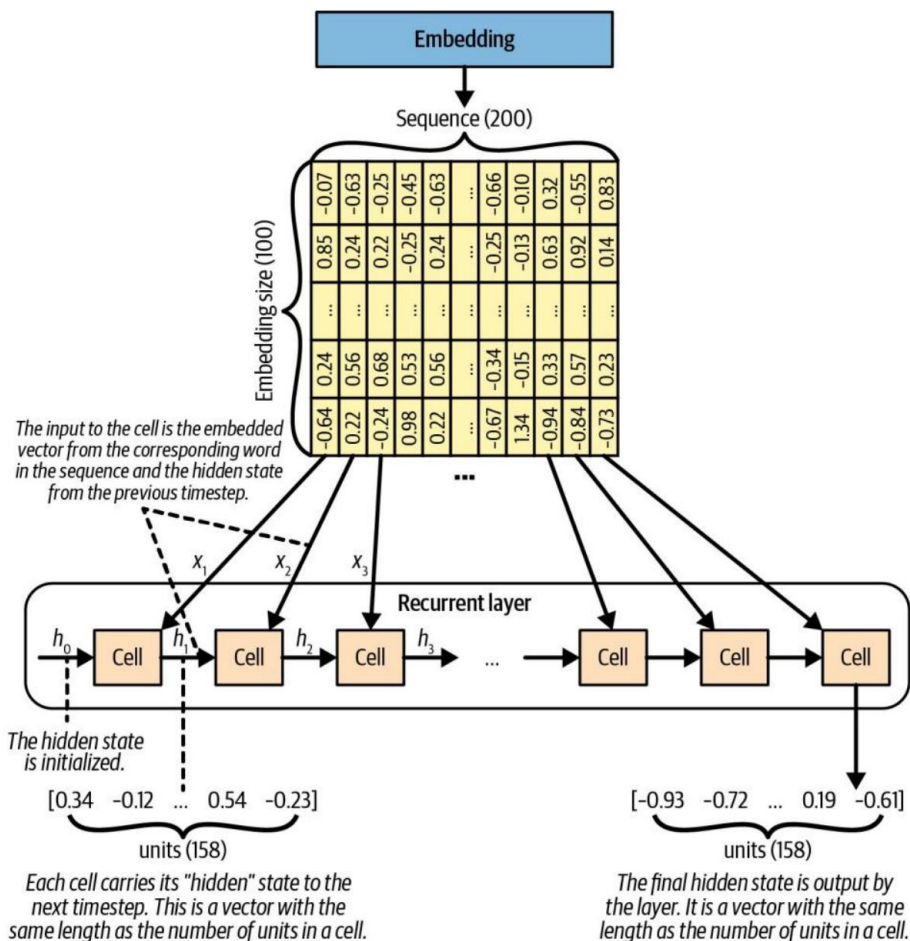


Figure 5-5. Comment une seule séquence traverse une couche récurrente

Ici, nous représentons le processus récurrent en dessinant une copie de la cellule à chaque pas de temps et montrons comment l'état caché est constamment mis à jour au fur et à mesure qu'il traverse les cellules. Nous pouvons clairement voir comment l'état caché précédent est mélangé avec le point de données séquentiel actuel (c'est-à-dire le vecteur de mots intégré actuel) pour produire l'état caché suivant. La sortie de la couche est l'état caché final de la cellule, après le traitement de chaque mot de la séquence d'entrée.



Le fait que la sortie de la cellule soit appelée état caché est une convention de dénomination malheureuse : elle n'est pas vraiment cachée et vous ne devriez pas la considérer comme telle. En effet, le dernier état caché est la sortie globale de la couche, et nous utiliserons le fait que nous pouvons accéder à l'état caché à chaque pas de temps plus loin dans ce chapitre.

La cellule LSTM

Maintenant que nous avons vu comment fonctionne une couche récurrente générique, examinons l'intérieur d'une cellule LSTM individuelle.

Le travail de la cellule LSTM est de générer un nouvel état caché, h_t , étant donné son état caché précédent, h_{t-1} , et l'incorporation de mots actuelle, x_t . Pour rappel, la longueur de h_t est égale au nombre d'unités dans le LSTM. Il s'agit d'un paramètre défini lorsque vous définissez le calque et n'a rien à voir avec la longueur de la séquence.



Assurez-vous de ne pas confondre le terme cellule avec unité. Il y a une cellule dans une couche LSTM qui est définie par le nombre d'unités qu'elle contient, de la même manière que la cellule de prisonnier de notre histoire précédente contenait de nombreux prisonniers. Nous dessinons souvent une couche récurrente sous la forme d'une chaîne de cellules déroulée, car cela permet de visualiser comment l'état caché est mis à jour à chaque pas de temps.

Une cellule LSTM maintient un état cellulaire, C_t , qui peut être considéré comme les croyances internes de la cellule concernant l'état actuel de la séquence. Ceci est distinct de l'état caché, h_t , qui est finalement généré par la cellule après le pas de temps final. L'état de la cellule a la même longueur que l'état caché (le nombre d'unités dans la cellule).

Examinons de plus près une seule cellule et comment l'état caché est mis à jour (Figure 5-6).

L'état masqué est mis à jour en six étapes :

1. L'état caché du pas de temps précédent, h_{t-1} , et le mot incorporé actuel, x_t , sont concaténés et passés par la porte d'oubli . Cette porte est simplement un dense

couche avec une matrice de poids W_f , un biais b_f et une fonction d'activation sigmoïde. Le vecteur résultant, f_t , a une longueur égale au nombre d'unités dans la cellule et contient des valeurs comprises entre 0 et 1 qui déterminent la part de l'état de cellule précédent, C_{t-1} , devraient être conservés.

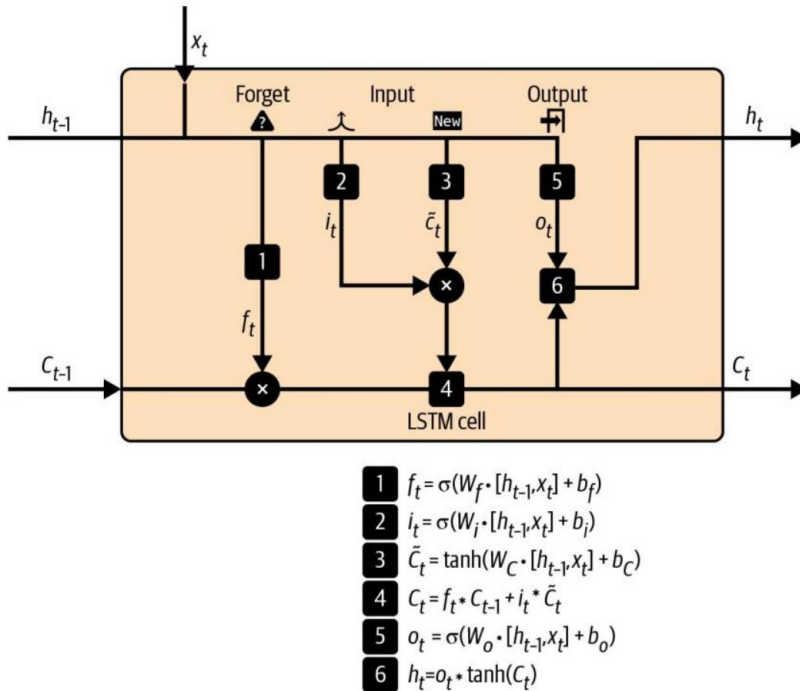


Figure 5-6. Une cellule LSTM

2. Le vecteur concaténé passe également par une porte d'entrée qui, comme la porte d'oubli, est une couche dense avec une matrice de poids W_i , un biais b_i et une fonction d'activation sigmoïde. La sortie de cette porte, elle, a une longueur égale au nombre d'unités dans la cellule et contient des valeurs comprises entre 0 et 1 qui déterminent la quantité de nouvelles informations qui seront ajoutées à l'état précédent de la cellule, C_t .
3. Le vecteur concaténé passe à travers une couche dense avec une matrice de poids W_C , un biais b_C et une fonction d'activation \tanh pour générer un vecteur \tilde{C}_t qui contient les nouvelles informations que la cellule souhaite envisager de conserver. Il a également une longueur égale à nombre d'unités dans la cellule et contient des valeurs comprises entre -1 et 1.

4. f_t et $C_t - 1$ sont multipliés par éléments et ajoutés au par élément
multiplication de celui-ci et C_t . Cela représente l'oubli de parties de l'état précédent de la cellule.
puis en ajoutant de nouvelles informations pertinentes pour produire l'état de cellule mis à jour, C_t .
5. Le vecteur concaténé passe à travers une porte de sortie : une couche dense avec une matrice de poids W_o , un biais b_o et une activation sigmoïde. Le vecteur résultant, o_t , a une longueur égale au nombre d'unités dans la cellule et stocke des valeurs comprises entre 0 et 1 qui déterminent la quantité d'état de cellule mis à jour, C_t , à sortir de la cellule.
6. o_t est multiplié élément par élément avec l'état de cellule mis à jour, C_t , après qu'une activation tanh ait été appliquée pour produire le nouvel état caché, h_t .



La couche Keras LSTM

Toute cette complexité est enveloppée dans le type de couche LSTM dans Keras, vous n'avez donc pas à vous soucier de l'implémenter vous-même !

Formation du LSTM

Le code pour construire, compiler et entraîner le LSTM est donné dans l'**exemple 5-8**. Exemple

5-8. Construire, compiler et former le LSTM

```
entrées = couches.Input(shape=(Aucun,), dtype="int32")
x = couches.Embedding(10000, 100)(entrées)
couches.LSTM(128, return_sequences=True)(x)
sorties =
layer.Dense(10000, activation = 'softmax')(x)
models.Model(entrées, sorties)

perte_fn =

loss.SparseCategoricalCrossentropy() lstm.compile("adam",
loss_fn) lstm.fit(train_ds, epochs=25)
```

- ❶ La couche d'entrée n'a pas besoin que nous spécifions la longueur de la séquence à l'avance (elle peut être flexible), nous utilisons donc None comme espace réservé.

- 2 La couche Embedding nécessite deux paramètres, la taille du vocabulaire (10 000 jetons) et la dimensionnalité du vecteur d'incorporation (100).
- 3 Les couches LSTM nous obligent à spécifier la dimensionnalité du vecteur caché (128). Nous choisissons également de renvoyer la séquence complète des états cachés, plutôt que simplement l'état caché au pas de temps final.
- 4 La couche Dense transforme les états cachés à chaque pas de temps en un vecteur de probabilités pour le jeton suivant.
- 5 Le modèle global prédit le prochain jeton, compte tenu d'une séquence d'entrée de jetons. Il le fait pour chaque jeton de la séquence.
- 6 Le modèle est compilé avec la perte SparseCategoricalCrossentropy : c'est la même chose que l'entropie croisée catégorielle, mais elle est utilisée lorsque les étiquettes sont des entiers plutôt que des vecteurs codés à chaud.
- 7 Le modèle est adapté à l'ensemble de données d'entraînement.

Dans la figure 5-7, vous pouvez voir les premières époques du processus de formation LSTM. Remarquez comment l'exemple de sortie devient plus compréhensible à mesure que la métrique de perte diminue. La figure 5-8 montre la métrique de perte d'entropie croisée tout au long de la formation.

processus.

```
Epoch 1/25
628/629 [=====>.] - ETA: 0s - loss: 4.4536
generated text:
recipe for mold salad are high 8 pickled to fold cook the dish into and warm in baking reduced but halves beans
and cut

629/629 [=====] - 29s 43ms/step - loss: 4.4527
Epoch 2/25
628/629 [=====>.] - ETA: 0s - loss: 3.2339
generated text:
recipe for racks - up-don with herb fizz | serve checking thighs onto sanding butter and baking surface in a hea
vy heavy large saucepan over blender ; stand overnight . [UNK] over moderate heat until very blended , garlic ,
about 8 minutes . cook airtight until cooked are soft seeds , about 1 45 minutes . sugar , until s is brown , 5
to sliced , parmesan , until browned and add extract . wooden crumb to outside of out sheets . flatten and prehe
ated return to the paste . add in pecans oval and let transfer day .

629/629 [=====] - 30s 48ms/step - loss: 3.2336
Epoch 3/25
629/629 [=====] - ETA: 0s - loss: 2.6229
generated text:
recipe for grilled chicken | preheat oven to 400*f . cook in large 8 - caramel grinder or until desired are firm
, about 6 minutes

629/629 [=====] - 27s 42ms/step - loss: 2.6229
Epoch 4/25
629/629 [=====] - ETA: 0s - loss: 2.3426
generated text:
recipe for pizza salad with sweet red pepper and star fruit | combine all ingredients except lowest ingredients
in a large skillet . working with batches and deglaze , cook until just cooked through , about 1 minute . meanwh
ile , boil potatoes and paprika in a little oil over medium - high heat , stirring it just until crisp , about 3
minutes . stir in bell pepper , onion and cooked paste and jalapeño until clams well after most - reggiano , abo
ut 5 minutes . transfer warm 2 tablespoons flesh of eggplants to medium bowl . serve .
```

Figure 5-7. Les premières époques du processus de formation LSTM

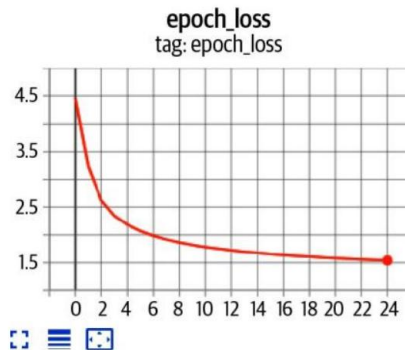


Figure 5-8. La métrique de perte d'entropie croisée du processus de formation LSTM par époque

Analyse du LSTM

Maintenant que nous avons compilé et entraîné le LSTM, nous pouvons commencer à l'utiliser pour générer de longues chaînes de texte en appliquant le processus suivant :

1. Alimenter le réseau avec une séquence de mots existante et demandez-lui de prédire le mot suivant.
2. Ajoutez ce mot à la séquence existante et répétez.

Le réseau produira un ensemble de probabilités pour chaque mot à partir duquel nous pouvons échantillonner. Par conséquent, nous pouvons rendre la génération de texte stochastique plutôt que déterministe. De plus, nous pouvons introduire un paramètre de température dans le processus d'échantillonnage pour indiquer dans quelle mesure nous souhaitons que le processus soit déterministe.



Le paramètre de température

Une température proche de 0 rend l'échantillonnage plus déterministe (c'est-à-dire que le mot avec la probabilité la plus élevée est très susceptible d'être choisi), tandis qu'une température de 1 signifie que chaque mot est choisi avec la probabilité fournie par le modèle.

Ceci est réalisé avec le code de l'[exemple 5-9](#), qui crée une fonction de rappel qui peut être utilisée pour générer du texte à la fin de chaque époque d'entraînement.

Exemple 5-9. La fonction de rappel TextGenerator

```
class TextGenerator(callbacks.Callback): def __init__(self,
index_to_word, top_k=10):

    self.index_to_word = index_to_word
    self.word_to_index = {
```

```

        mot : index pour index, mot dans
    énumérer (index_to_word) } ❶

    def sample_from(self, probs, température): ❷
        probs = probs ** (1 /
température)
        problèmes = problèmes /
np.sum (prob)
        retourner np.random.choice(len(probs), p=probs), probs

    def generate(self, start_prompt, max_tokens, température): start_tokens =
[
    self.word_to_index.get(x, 1) pour x dans start_prompt.split()
] ❸
    sample_token = Aucune info = [] while len(start_tokens) <
max_tokens et sample_token !=
0 : ❹
        x = np.array([start_tokens])
    y = self.model.predict(x) sample_token, ❺
        probs = self.sample_from(y[0][-1],
température) ❻
        info.append({'prompt' : start_prompt ,
'word_probs' : problèmes})
        start_tokens.append (sample_token) start_prompt =
start_prompt +
self.index_to_word[sample_token]
        print(f"\n texte généré :
\n(start_prompt)\n")
        renvoyer des informations

    déf on_epoch_end(soi, époque, logs=Aucun) :
self.generate("recette pour", max_tokens = 100, température =

```

❶ 1.0) Créer un mappage de vocabulaire inverse (du mot au jeton).

❷ Cette fonction met à jour les probabilités avec un facteur d'échelle de température .

❸ L'invite de démarrage est une chaîne de mots que vous souhaitez donner au modèle pour démarrer le processus de génération (par exemple, recette). Les mots sont d'abord convertis en une liste de jetons.

❹ La séquence est générée jusqu'à ce qu'elle soit longue de max_tokens ou qu'un jeton d'arrêt (0) soit produit.

❺ Le modèle génère les probabilités que chaque mot soit le prochain dans la séquence.

❻ Les probabilités passent par l'échantillonneur pour sortir le mot suivant, paramétré par la température.

❼

Nous ajoutons le nouveau mot au texte d'invite, prêt pour la prochaine itération du processus génératif.

Jetons un coup d'œil à cela en action, à deux valeurs de température différentes (Figure 5-9).

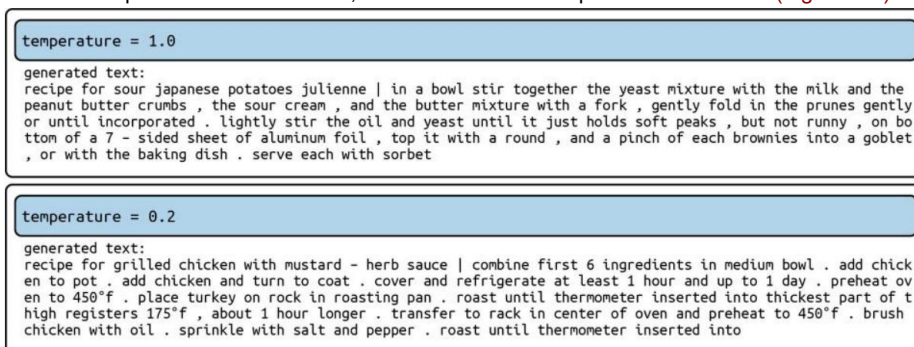


Figure 5-9. Sorties générées à température = 1,0 et température = 0,2

Il y a quelques points à noter à propos de ces deux passages. Premièrement, les deux sont stylistiquement similaires à une recette du kit de formation original. Ils s'ouvrent tous deux sur un titre de recette et contiennent des constructions généralement grammaticalement correctes. La différence est que le texte généré avec une température de 1,0 est plus aventureux et donc moins précis que l'exemple avec une température de 0,2. La génération de plusieurs échantillons avec une température de 1,0 entraînera donc une plus grande variété, car le modèle échantillonne à partir d'une distribution de probabilité avec une plus grande variance.

Pour le démontrer, la figure 5-10 montre les cinq premiers jetons présentant les probabilités les plus élevées pour une plage d'invites, pour les deux valeurs de température.

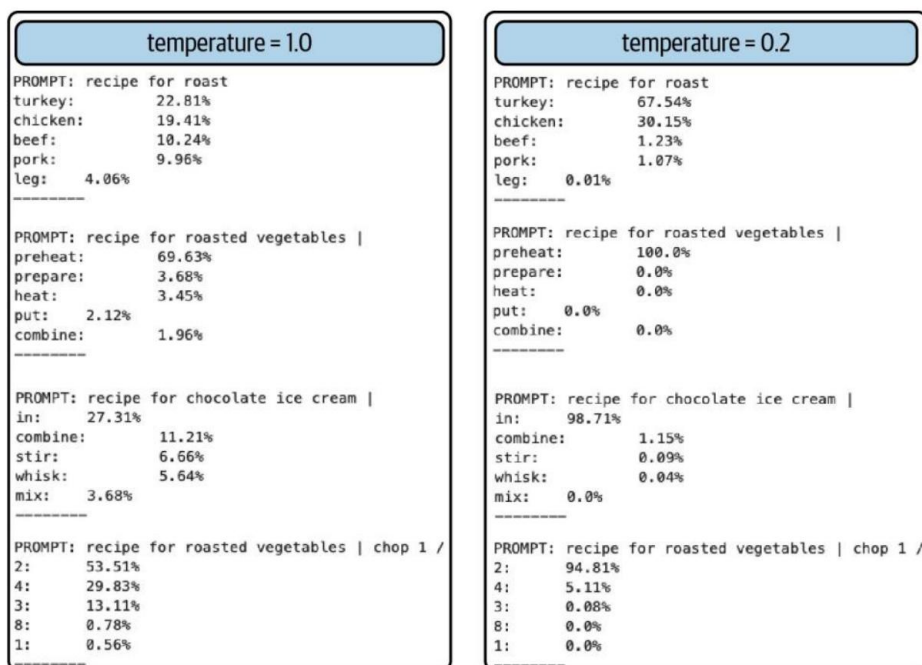


Figure 5-10. Répartition des probabilités de mots suivant diverses séquences, pour des valeurs de température de 1,0 et 0,2

Le modèle est capable de générer une distribution appropriée pour le mot suivant le plus probable dans une gamme de contextes. Par exemple, même si le modèle n'a jamais été informé des parties du discours telles que les noms, les verbes ou les nombres, il est généralement capable de séparer les mots dans ces classes et utilise-les d'une manière grammaticalement correcte.

De plus, le modèle est capable de sélectionner un verbe approprié pour commencer les instructions de la recette, en fonction du titre précédent. Pour les légumes rôtis, il sélectionne préchauffer, préparer, chauffer, mettre ou combiner comme possibilités les plus probables, tandis que pour la crème glacée, il sélectionne , combine, remue, fouette et mélange. Cela montre que le modèle possède une certaine compréhension contextuelle des différences entre les recettes en fonction de leurs ingrédients.

Notez également comment les probabilités pour les exemples de température = 0,2 sont beaucoup plus fortement pondérées en faveur du jeton de premier choix. C'est la raison pour laquelle il y a généralement moins de variété dans les générations lorsque la température est plus basse.

Bien que notre modèle LSTM de base fasse un excellent travail pour générer un texte réaliste, il est clair qu'il a encore du mal à saisir une partie de la signification sémantique des mots qu'il génère. Il introduit des ingrédients qui ne fonctionneront probablement pas bien ensemble (par exemple, des pommes de terre japonaises aigres, des miettes de noix de pécan et un sorbet) ! Dans certains cas, cela peut être souhaitable (par exemple, si nous voulons que notre LSTM génère des modèles de mots intéressants et uniques), mais dans d'autres cas, nous aurons besoin de notre modèle pour mieux comprendre la manière dont les mots peuvent être regroupés. et une mémoire plus longue des idées introduites plus tôt dans le texte.

Dans la section suivante, nous explorerons certaines des façons dont nous pouvons améliorer notre réseau LSTM de base. Au [chapitre 9](#), nous examinerons un nouveau type de modèle autorégressif, le Transformer, qui fait passer la modélisation du langage à un niveau supérieur.

Extensions de réseaux neuronaux récurrents (RNN)

Le modèle de la section précédente est un exemple simple de la façon dont un LSTM peut être formé pour apprendre à générer du texte dans un style donné. Dans cette section, nous explorerons plusieurs extensions de cette idée.

Réseaux récurrents empilés

Le réseau que nous venons d'examiner contenait une seule couche LSTM, mais nous pouvons également entraîner des réseaux avec des couches LSTM empilées, afin que des fonctionnalités plus approfondies puissent être apprises du texte.

Pour y parvenir, nous introduisons simplement une autre couche LSTM après la première. La deuxième couche LSTM peut ensuite utiliser les états cachés de la première couche comme données d'entrée. Ceci est illustré dans [la figure 5-11](#) et l'architecture globale du modèle est présentée dans [le tableau 5-2](#).

Extensions de réseaux neuronaux récurrents (RNN)

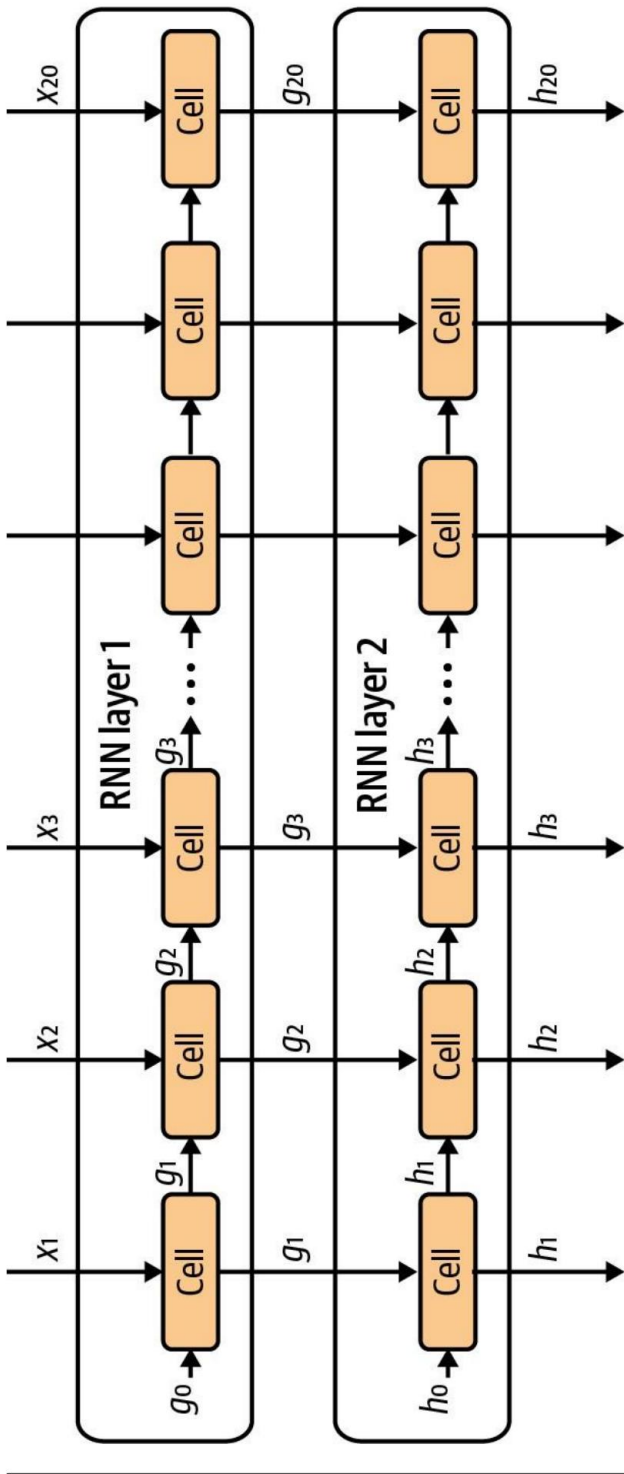


Figure 5-11. Diagram of a multilayer RNN: g denotes hidden states of the first layer and h denotes hidden states of the second layer

Tableau 5-2. Résumé du modèle du LSTM empilé

Sortie de couche (type)	Paramètre #
forme	
InputLayer (Aucun, Aucun)	0
Incorporation (Aucun, Aucun, 100)	1 000 000
LSTM (Aucun, Aucun, 128)	117 248
LSTM (Aucun, Aucun, 128)	131 584
Dense (Aucun, Aucun, 10 000)	1 290 000
Paramètres totaux	2 538 832
Paramètres entraîables	2 538 832
Paramètres non entraîables	0

Le code pour construire le LSTM empilé est donné dans l'exemple 5-10.

Exemple 5-10. Construire un LSTM empilé

```
text_in = layer.Input(shape = (Aucun,))
embedding = layer.Embedding(total_words, embedding_size)(text_in)
x = couches.LSTM (n_units, return_sequences =
Vrai)(x) x = couches.LSTM(n_units, return_sequences
= Vrai)(x)
probabilités = couches.Dense (total_words, activation =
'softmax')(x) modèles.Modèle(text_in,
probabilités)
```

Unités récurrentes fermées

Un autre type de couche RNN couramment utilisé est l'unité récurrente fermée (GRU).² Les principales différences par rapport à l'unité LSTM sont les suivantes :

1. Les portes d'oubli et d'entrée sont remplacées par des portes de réinitialisation et de mise à jour .
2. Il n'y a pas d'état de cellule ni de porte de sortie , seulement un état caché qui est émis par le cellule.

L'état masqué est mis à jour en quatre étapes, comme illustré dans la figure 5-12.

Extensions de réseaux neuronaux récurrents (RNN)

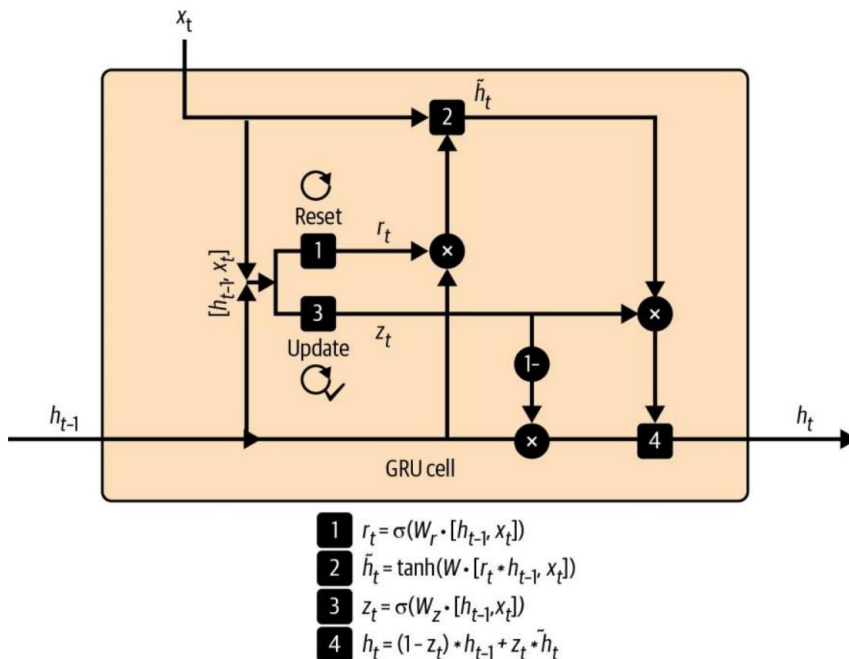


Figure 5-12. Une seule cellule GRU

Le processus est le suivant :

1. L'état caché du pas de temps précédent, h_{t-1} , et l'intégration de mots actuelle, x_t , sont concaténés et utilisés pour créer la porte de réinitialisation. Cette porte est une couche dense, avec une matrice de poids W_r et une fonction d'activation sigmoïde. Le vecteur résultant, r_t , a une longueur égale au nombre d'unités dans la cellule et stocke des valeurs comprises entre 0 et 1 qui déterminent la quantité de l'état caché précédent, h_{t-1} , qui doit être transportée. avancer dans le calcul des nouvelles croyances de la cellule.
2. La porte de réinitialisation est appliquée à l'état caché, h_{t-1} , et concaténée avec l'incorporation de mots actuelle, x_t . Ce vecteur est ensuite introduit dans une couche dense avec une matrice de poids W et une fonction d'activation \tanh pour générer un vecteur, \tilde{h}_t , qui stocke les nouvelles croyances de la cellule. Sa longueur est égale au nombre d'unités dans la cellule et stocke les valeurs comprises entre -1 et 1.
3. La concaténation de l'état caché du pas de temps précédent, h_{t-1} , et du mot d'intégration actuel, x_t , sont également utilisées pour créer la porte de mise à jour. Cette porte est un

couche dense avec une matrice de poids W_z et une activation sigmoïde. Le vecteur résultant, z_t , a une longueur égale au nombre d'unités dans la cellule et stocke les valeurs

entre 0 et 1, qui sont utilisés pour déterminer la quantité de nouvelles croyances, h_t , à se fondre dans l'état caché \tilde{h}_t actuel, $h_t - 1$.

4. Les nouvelles croyances de la cellule, \tilde{h}_t , et l'état caché actuel, $h_t - 1$, sont mélangés dans une proportion déterminée par la porte de mise à jour, z_t , pour produire l'état caché mis à jour, h_t , qui est généré par la cellule.

Cellules bidirectionnelles

Pour les problèmes de prédiction où le texte entier est disponible pour le modèle au moment de l'inférence, il n'y a aucune raison de traiter la séquence uniquement dans le sens avant : elle pourrait tout aussi bien être traitée vers l'arrière. Une couche bidirectionnelle profite de cela en stockant deux ensembles d'états cachés : un qui est produit à la suite du traitement de la séquence dans le sens habituel vers l'avant et un autre qui est produit lorsque la séquence est traitée vers l'arrière. De cette façon, la couche peut apprendre des informations précédant et suivant le pas de temps donné.

Dans Keras, ceci est implémenté comme un wrapper autour d'une couche récurrente, comme le montre l' [exemple 5-11](#).

Exemple 5-11. Construire une couche GRU bidirectionnelle

```
couche = couches.Bidirectionnel(couches.GRU(100))
```



État caché

Les états cachés dans la couche résultante sont des vecteurs de longueur égale au double du nombre d'unités dans la cellule encapsulée (une concaténation des états cachés avant et arrière). Ainsi, dans cet exemple les états cachés de la couche sont des vecteurs de longueur 200.

Jusqu'à présent, nous n'avons appliqué que des modèles autorégressifs (LSTM) aux données textuelles. Dans la section suivante, nous verrons comment les modèles autorégressifs peuvent également être utilisés pour générer des images.

PixelCNN

En 2016, van den Oord et al.³ ont introduit un modèle qui génère des images pixel par pixel en prédisant la probabilité du pixel suivant en fonction des pixels qui le précèdent. Le modèle s'appelle PixelCNN et peut être entraîné pour générer des images de manière autorégressive. Il y a deux nouveaux concepts que nous devons introduire pour comprendre le PixelCNN : les couches convolutives masquées et les blocs résiduels.



Exécution du code pour cet exemple

Le code de cet exemple se trouve dans le notebook Jupyter situé dans `notebooks/05_autoregressive/02_pixelcnn/pixelcnn.ipynb` dans le référentiel de livres.

Le code a été adapté de l'excellent [tutoriel PixelCNN](#) créé par ADMoreau, disponible sur le site de Keras.

Couches convolutives masquées

Comme nous l'avons vu au [chapitre 2](#), une couche convolutive peut être utilisée pour extraire des caractéristiques d'une image en appliquant une série de filtres. La sortie de la couche sur un pixel particulier est une somme pondérée des poids de filtre multipliée par les valeurs de la couche précédente sur un petit carré centré sur le pixel. Cette méthode peut détecter les bords et les textures et, au niveau des couches plus profondes, les formes et les caractéristiques de niveau supérieur.

Bien que les couches convolutives soient extrêmement utiles pour la détection de caractéristiques, elles ne peuvent pas être utilisées directement dans un sens autorégressif, car aucun ordre n'est placé sur les pixels. Ils reposent sur le fait que tous les pixels sont traités de la même manière : aucun pixel n'est traité comme le début ou la fin de l'image. Cela contraste avec les données textuelles que nous avons déjà vues dans ce chapitre, où les jetons sont clairement ordonnés afin que les modèles récurrents tels que les LSTM puissent être facilement appliqués.

Pour que nous puissions appliquer des couches convolutives à la génération d'images dans un sens autorégressif, nous devons d'abord placer un ordre sur les pixels et nous assurer que les filtres ne peuvent voir que les pixels qui précèdent le pixel en question. Nous pouvons ensuite générer des images pixel par pixel, en appliquant des filtres convolutifs à l'image actuelle pour prédire la valeur du pixel suivant à partir de tous les pixels précédents.

Nous devons d'abord choisir un ordre pour les pixels. Une suggestion judicieuse est d'ordonner les pixels du haut à gauche vers le bas à droite, en se déplaçant d'abord le long des lignes, puis vers le bas des colonnes.

Nous masquons ensuite les filtres convolutifs afin que la sortie du calque à chaque pixel ne soit influencée que par les valeurs de pixel qui précèdent le pixel en question. Ceci est obtenu en multipliant un masque de uns et de zéros avec la matrice de poids de filtre, de sorte que les valeurs de tous les pixels situés après le pixel cible soient remises à zéro.

Il existe en fait deux types différents de masques dans un PixelCNN, comme le montre la [figure 5](#).

13 :

- Type A, où la valeur du pixel central est masquée
- Type B, où la valeur du pixel central n'est pas masquée

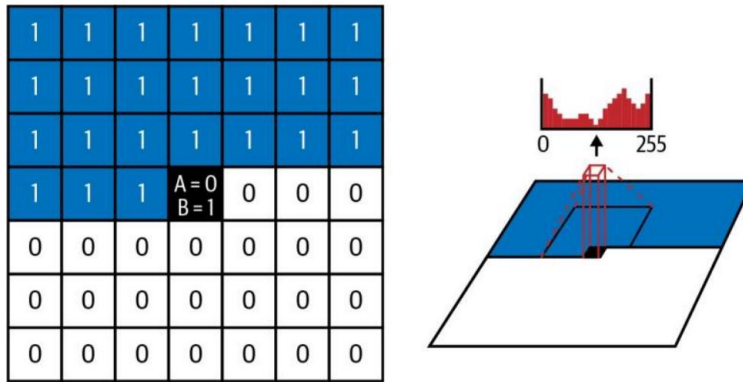


Figure 5-13. À gauche : un masque filtrant convolutif ; à droite : un masque appliqué à un ensemble de pixels pour prédire la distribution de la valeur du pixel central (source : [van den Oord et al., 2016](#))

La couche convolutive masquée initiale (c'est-à-dire celle qui est appliquée directement à l'image d'entrée) ne peut pas utiliser le pixel central, car c'est précisément le pixel que nous voulons que le réseau devine ! Cependant, les couches suivantes peuvent utiliser le pixel central car celui-ci aura été calculé uniquement à partir des informations des pixels précédents dans l'image d'entrée d'origine.

Nous pouvons voir dans l'[exemple 5-12](#) comment un `MaskedConvLayer` peut être construit à l'aide de Keras.

Exemple 5-12. Un `MaskedConvLayer` dans Keras

```
class MaskedConvLayer(layers.Layer):
    def __init__(self, mask_type, **kwargs):
        super(MaskedConvLayer, self).__init__()
        self.mask_type = mask_type

    self.conv = layer.Conv2D(**kwargs)

    def build(self, auto, input_shape):
        self.conv.build(input_shape)
        forme_noyau = self.conv.kernel.get_shape()
        np.zeros(shape=kernel_shape)
        kernel_shape[0] // 2, ...] = 1.0
        self.mask[kernel_shape[0] // 2, : kernel_shape[1] // 2, ...] = 1.0
        si self.mask_type == "B":
            self.mask[kernel_shape[0] // 2, kernel_shape[1] // 2, ...] = 1.0
```

1

2

3

4

5

```
appel def (auto, entrées):
self.conv.kernel.assign(self.conv.kernel * self.mask) return self.conv(inputs) ❸
```

- ❶ Le MaskedConvLayer est basé sur la couche Conv2D normale .
- ❷ Le masque est initialisé avec tous des zéros.
- ❸ Les pixels des lignes précédentes sont démasqués avec des uns.
- ❹ Les pixels des colonnes précédentes qui se trouvent sur la même ligne sont démasqués avec ceux.
- ❺ Si le type de masque est B, le pixel central est démasqué avec un un.
- ❻ Le masque est multiplié par les poids du filtre.

Notez que cet exemple simplifié suppose une image en niveaux de gris (c'est-à-dire avec un seul canal). Si nous avons des images en couleur, nous aurons trois canaux de couleur sur lesquels nous pourrions également placer un ordre de sorte que, par exemple, le canal rouge précède le canal bleu, qui précède le canal vert.

Blocs résiduels

Maintenant que nous avons vu comment masquer la couche convolutive, nous pouvons commencer à construire notre PixelCNN. Le bloc de base que nous utiliserons est le bloc résiduel.

Un bloc résiduel est un ensemble de couches où la sortie est ajoutée à l'entrée avant d'être transmise au reste du réseau. En d'autres termes, l'entrée dispose d'un chemin rapide vers la sortie, sans avoir à passer par les couches intermédiaires : c'est ce qu'on appelle une connexion sautée. La raison derrière l'inclusion d'une connexion sautée est que si la transformation optimale consiste simplement à conserver la même entrée, cela peut être réalisé en remettant simplement à zéro les poids des couches intermédiaires. Sans la connexion sautée, le réseau devrait trouver un mappage d'identité à travers les couches intermédiaires, ce qui est beaucoup plus difficile.

Un diagramme du bloc résiduel dans notre PixelCNN est présenté à la [figure 5-14](#).

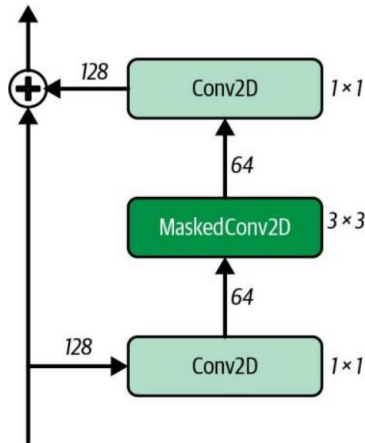


Figure 5-14. Un bloc résiduel PixelCNN (les numéros de filtres sont à côté des flèches et les tailles de filtres sont à côté des calques)

Nous pouvons construire un ResidualBlock en utilisant le code présenté dans l'exemple 5-13.

Exemple 5-13. Un bloc résiduel

```

classe ResidualBlock(layers.Layer) : __init__(self, filtres,          déf
**kwargs) : super(ResidualBlock, self).__init__(**kwargs)

    self.conv1 = couches.Conv2D(
        filtres=filtres // 2, kernel_size=1, activation="relu"
    )
    self.pixel_conv = MaskedConv2D (
        mask_type="B",
        filtres=filtres // 2, kernel_size=3,
        activation="relu",
        padding="same",
    )
    self.conv2 = couches.Conv2D(
        filtres=filtres, kernel_size=1, activation="relu" )

    def call (self, entrées): = self.conv1 (entrées)          x
    self.pixel_conv (x) self.conv2 (x)          X =
          X =
    retour
    layer.add([entrées, x])

```

❶ La couche Conv2D initiale réduit de moitié le nombre de canaux.

- ❷ La couche MaskedConv2D de type B avec une taille de noyau de 3 utilise uniquement les informations de cinq pixels : trois pixels dans la ligne au-dessus du pixel de focus, un à gauche et le pixel de focus lui-même.
- ❸ La couche Conv2D finale double le nombre de canaux pour correspondre à nouveau à la forme d'entrée.
- ❹ La sortie des couches convolutives est ajoutée à l'entrée : c'est la connexion sautée.

Entraîner le PixelCNN

Dans l'exemple 5-14, nous avons rassemblé l'ensemble du réseau PixelCNN, en suivant approximativement la structure présentée dans l'article original. Dans le document original, la couche de sortie est une couche Conv2D à 256 filtres, avec activation softmax. En d'autres termes, le réseau tente de recréer son entrée en prédisant les valeurs correctes des pixels, un peu à la manière d'un auto-encodeur. La différence est que le PixelCNN est contraint de sorte qu'aucune information provenant des pixels précédents ne puisse circuler pour influencer la prédiction de chaque pixel, en raison de la façon dont le réseau est conçu, à l'aide de couches MaskedConv2D.

Un défi avec cette approche est que le réseau n'a aucun moyen de comprendre qu'une valeur de pixel de, disons, 200 est très proche d'une valeur de pixel de 201. Il doit apprendre chaque valeur de sortie de pixel indépendamment, ce qui signifie que la formation peut être très lente. même pour les ensembles de données les plus simples. Par conséquent, dans notre implémentation, nous simplifions plutôt la saisie afin que chaque pixel ne puisse prendre qu'une seule valeur parmi quatre. De cette façon, nous pouvons utiliser une couche de sortie Conv2D à 4 filtres au lieu de 256.

Exemple 5-14. L'architecture PixelCNN

```
entrées = couches.Input(shape=(16, 16, 1))
x =
MaskedConv2D(mask_type="A"
, filters=128
, taille_noyau=7
, activation="relu"
, padding="même")(entrées)

pour _ dans la plage (5):
    x = Bloc Résiduel (filtres = 128) (x)

pour _ dans la plage (2) :
    x = MaskedConv2D
( mask_type="B",
filters=128,
```

```

kernel_size=1,
strides=1,
activation="relu",
padding="valid",
)(X) ④

out = couches.Conv2D(
    filters=4, kernel_size=1, strides=1, activation="softmax",

padding="valid" )(x) pixel_cnn = models.Model(entrées, sortie) ⑥

adam = optimiseurs.Adam(learning_rate=0.0005)
pixel_cnn.compile(optimizer=adam,
    loss="sparse_categorical_crossentropy")

pixel_cnn.fit ( input_data

    , des données de sortie
    , taille_lot = 128
    , époques=150
) ⑦

```

- ① L'entrée du modèle est une image en niveaux de gris de taille $16 \times 16 \times 1$, avec des entrées mises à l'échelle entre 0 et 1.
- ② La première couche MaskedConv2D de type A avec une taille de noyau de 7 utilise les informations de 24 pixels : 21 pixels dans les trois lignes au-dessus du pixel de focus et 3 à gauche (le pixel de focus lui-même n'est pas utilisé).
- ③ Cinq groupes de couches ResidualBlock sont empilés séquentiellement.
- ④ Deux couches MaskedConv2D de type B avec une taille de noyau de 1 agissent comme des couches denses sur le nombre de canaux pour chaque pixel.
- ⑤ La couche Conv2D finale réduit le nombre de canaux à quatre, soit le nombre de niveaux de pixels pour cet exemple.
- ⑥ Le modèle est conçu pour accepter une image et générer une image des mêmes dimensions.
- ⑦ Ajuster le modèle : input_data est mis à l'échelle dans la plage [0, 1] (flotteurs) ; des données de sortie est mis à l'échelle dans la plage [0, 3] (entiers).

Analyse du PixelCNN

Nous pouvons entraîner notre PixelCNN sur des images de l'ensemble de données Fashion-MNIST que nous avons rencontré au [chapitre 3](#). Pour générer de nouvelles images, nous devons demander au modèle de prédire le pixel suivant en fonction de tous les pixels précédents, un pixel à la fois. C'est un processus très lent par rapport à un modèle tel qu'un auto-encodeur variationnel ! Pour une image en niveaux de gris 32×32 , nous devons faire 1 024 prédictions séquentiellement à l'aide du modèle, par rapport à la seule prédiction que nous devons faire pour un VAE. C'est l'un des principaux inconvénients des modèles autorégressifs tels que PixelCNN : ils sont lents à échantillonner, en raison de la nature séquentielle du processus d'échantillonnage.

Pour cette raison, nous utilisons une taille d'image de 16×16 , plutôt que de 32×32 , pour accélérer la génération de nouvelles images. La classe de rappel de génération est présentée dans [l'exemple 5-15](#).

Exemple 5-15. Générer de nouvelles images à l'aide de PixelCNN

```

classe
ImageGenerator(callbacks.Callback) : def __init__(self, num_img) :
    self.num_img = num_img

    def sample_from(self, probs, température): (1 /
    température) / np.sum                problèmes = problèmes **
    (probs)                               problèmes = problèmes

    retourner np.random.choice(len(probs), p=probs)

    def generate(self, température) : generate_images =
    np.zeros(
        shape=(self.num_img,) + (pixel_cnn.input_shape)[1:]
    ) ❶
    lot, lignes, colonnes, canaux = generate_images.shape

    pour la ligne dans la plage (lignes): plage                pour col dedans
    (cols): plage (canaux):                pour le canal dans
    probs =
    self.model.predict(generated_images)[
        :, rangée, col, :
    ] ❷
    images_générées[:, ligne, col, canal] = [
    self.sample_from(x, température) pour x dans les problèmes
    ] ❸
    images_générées[:, ligne, col, canal]
    /= 4 ❹
    renvoyer les images_générées

    def on_epoch_end(self, epoch, logs=None) : generate_images =
    self.generate(temperature =
    1.0)

    display( images_générées,

```

```

save_to = "/output/generated_img_%03d.png" % (époque)
s)      img_generator_callback

```

Générateur d'images (num_img = 10)

❶ Commencez par un

lot d'images vides (tous des zéros).

- ❷ Parcourez les lignes, les colonnes et les canaux de l'image actuelle, en prédisant la distribution de la valeur de pixel suivante.
- ❸ Échantillonnez un niveau de pixel à partir de la distribution prédite (pour notre exemple, un niveau compris dans la plage [0, 3]).
- ❹ Convertissez le niveau de pixel dans la plage [0, 1] et écrasez la valeur du pixel dans le image actuelle, prête pour la prochaine itération de la boucle.

Dans la figure 5-15, nous pouvons voir plusieurs images de l'ensemble de formation d'origine, ainsi que des images générées par PixelCNN.



Figure 5-15. Exemples d'images de l'ensemble de formation et d'images générées créées par le modèle PixelCNN

Le modèle fait un excellent travail en recréant la forme générale et le style des images originales ! Il est assez étonnant que nous puissions traiter les images comme une série de jetons (valeurs de pixels) et appliquer des modèles autorégressifs tels que PixelCNN pour produire des échantillons réalistes.

Comme mentionné précédemment, l'un des inconvénients des modèles autorégressifs est qu'ils sont lents à échantillonner, c'est pourquoi un exemple simple de leur application est présenté dans ce livre. Cependant, comme nous le verrons au [chapitre 10](#), des formes plus complexes de modèles autorégressifs peuvent être appliquées aux images pour produire des résultats de pointe. Dans de tels cas, la lenteur de la génération est un prix nécessaire à payer en échange de résultats de qualité exceptionnelle.

Depuis la publication de l'article original, plusieurs améliorations ont été apportées à l'architecture et au processus de formation de PixelCNN. La section suivante présente l'un de ces changements (à l'aide de distributions de mélange) et montre comment entraîner un modèle PixelCNN avec cette amélioration à l'aide d'une fonction TensorFlow intégrée.

Distributions de mélanges

Pour notre exemple précédent, nous avons réduit la sortie de PixelCNN à seulement 4 niveaux de pixels pour garantir que le réseau n'ait pas besoin d'apprendre une distribution sur 256 valeurs de pixels indépendantes, ce qui ralentirait le processus de formation. Cependant, c'est loin d'être idéal : pour les images couleur, nous ne voudrions pas que notre toile soit limitée à seulement une poignée de couleurs possibles.

Pour contourner ce problème, nous pouvons faire en sorte que la sortie du réseau soit une distribution de mélange, au lieu d'un softmax sur 256 valeurs de pixels discrets, en suivant les idées présentées par Salimans et al. Une distribution de mélange est tout simplement un mélange de deux ou plusieurs d'autres distributions de probabilité. Par exemple, nous pourrions avoir une distribution mixte de cinq distributions logistiques, chacune avec des paramètres différents. La distribution mixte nécessite également une distribution catégorielle discrète qui dénote la probabilité de choisir chacune des distributions incluses dans le mélange. Un exemple est présenté dans [la Figure 5-16](#).

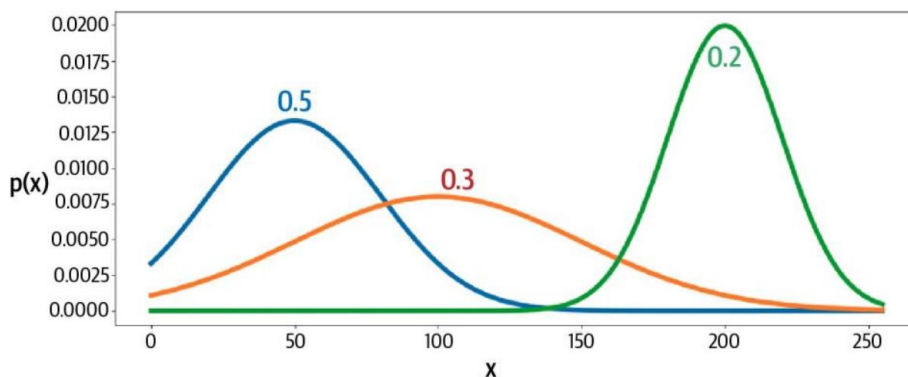


Figure 5-16. Une distribution mixte de trois distributions normales avec des paramètres différents : la distribution catégorielle sur les trois distributions normales est [0,5, 0,3, 0,2]

Pour échantillonner à partir d'une distribution mixte, nous échantillonons d'abord à partir de la distribution catégorielle pour choisir une sous-distribution particulière, puis nous échantillonons à partir de celle-ci de la manière habituelle. De cette façon, nous pouvons créer des distributions complexes avec relativement peu de paramètres.

Par exemple, la distribution du mélange de la figure 5-16 ne nécessite que huit paramètres -deux pour la distribution catégorielle et une moyenne et une variance pour chacune des trois distributions normales. Ceci est comparé aux 255 paramètres qui définiraient une distribution catégorielle sur toute la plage de pixels.

Idéalement, la bibliothèque TensorFlow Probability fournit une fonction qui nous permet de créer un PixelCNN avec une sortie de distribution de mélange sur une seule ligne. L'exemple 5-16 illustre comment créer un PixelCNN à l'aide de cette fonction.



Exécution du code pour cet exemple

Le code de cet exemple se trouve dans le notebook Jupyter dans `notebooks/05_autoregressive/03_pixelcnn_md/pixelcnn_md.ipynb` dans le référentiel de livres.

Exemple 5-16. Construire un PixelCNN à l'aide de la fonction TensorFlow

```
importer tensorflow_probability en tant que tfp
```

```
dist = tfp.distributions.PixelCNN( image_shape=(32,
32, 1),
    num_resnet=1,
    num_hierarchies=2,          num_filters=32,
    num_logistic_mix=5,        dropout_p=.3, )
```

```
❶ image_input = layer.Input(shape=(32,
```

```
32, 1)) log_prob =
```

```
dist.log_prob(image_input)
```

```
model = models.Model(inputs=image_input, sorties=log_prob)
```

```
model.add_loss(-tf.reduce_mean(log_prob))
```

❸

❹

- ❶ Définissez le PixelCNN comme une distribution, c'est-à-dire que la couche de sortie est une distribution mixte composée de cinq distributions logistiques. L'entrée est une image en niveaux de gris de taille $32 \times 32 \times 1$.
- ❷ Le modèle prend une image en niveaux de gris en entrée et génère la log-vraisemblance de l'image sous la distribution de mélange calculée par PixelCNN.
- ❸ La fonction de perte est la log-vraisemblance négative moyenne sur le lot d'images d'entrée.

Le modèle est entraîné de la même manière que précédemment, mais cette fois en acceptant des pixels entiers valeurs en entrée, dans la plage $[0, 255]$. Les sorties peuvent être générées à partir de la distribution en utilisant la fonction exemple , comme indiqué dans l'exemple 5-17. Exemple 5-17. Échantillonnage de la distribution du mélange PixelCNN `dist.sample(10).numpy()`

Des exemples d'images générées sont présentés dans la figure 5-17. La différence par rapport à nos exemples précédents est que désormais toute la plage des valeurs de pixels est utilisée.



Figure 5-17. Sorties du PixelCNN utilisant une sortie de distribution de mélange

Résumé

Dans ce chapitre, nous avons vu comment des modèles autorégressifs tels que les réseaux de neurones récurrents peuvent être appliqués pour générer des séquences de texte imitant un style d'écriture particulier, et également comment un PixelCNN peut générer des images de manière séquentielle, un pixel à la fois.

Nous avons exploré deux types différents de couches récurrentes : la mémoire à long terme (LSTM) et l'unité récurrente fermée (GRU) - et avons vu comment ces cellules peuvent être empilées ou rendues bidirectionnelles pour former des architectures de réseau plus complexes. Nous avons construit un LSTM pour générer des recettes réalistes à l'aide de Keras et avons vu comment manipuler la température de

le processus d'échantillonnage pour augmenter ou diminuer le caractère aléatoire de la sortie. Nous avons également vu comment des images peuvent être générées de manière autorégressive, à l'aide d'un PixelCNN. Nous avons construit un PixelCNN à partir de zéro en utilisant Keras, en codant les couches convolutives masquées et les blocs résiduels pour permettre aux informations de circuler à travers le réseau afin que seuls les pixels précédents puissent être utilisés pour générer le pixel actuel. Enfin, nous avons expliqué comment la bibliothèque TensorFlow Probability fournit une fonction PixelCNN autonome qui implémente une distribution de mélange comme couche de sortie, nous permettant d'améliorer encore le processus d'apprentissage.

Dans le chapitre suivant, nous explorerons une autre famille de modélisation générative qui modélise explicitement la distribution génératrice de données : les modèles de flux normalisants. Les références

1. Sepp Hochreiter et Jürgen Schmidhuber, « Mémoire à long terme », Calcul neuronal 9 (1997) : 1735-1780, <https://www.bioinf.jku.at/publications/older/2604.pdf> .
2. Kyunghyun Cho et al., « Apprentissage des représentations d'expressions à l'aide d'un encodeur-décodeur RNN pour Traduction automatique statistique », 3 juin 2014, <https://arxiv.org/abs/1406.1078> .
3. Aaron van den Oord et al., « Réseaux de neurones récurrents de pixels », 19 août 2016, <https://arxiv.org/abs/1601.06759>.
4. Tim Salimans et al., « PixelCNN++ : améliorer le PixelCNN avec un mélange logistique discret Probabilité et autres modifications », 19 janvier 2017, <http://arxiv.org/abs/1701.05517> .