Université de Montréal

Préentraînement d'un modèle ELECTRA

par

Simon Théorêt

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Rapport de stage présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique,

7 février 2025

Université de Montréal

Faculté des arts et des sciences

Ce rapport de stage intitulé

Préentraînement d'un modèle ELECTRA

présenté par

Simon Théorêt

a été évalué par un jury composé des personnes suivantes :

Nom du président du jury
(président-rapporteur)

Nom du directeur de recherche
(directeur de recherche)

Nom du membre de jury
(membre du jury)

Résumé

Le logiciel Antidote permet de corriger des textes en français et en anglais. Il détecte plusieurs milliers de types d'erreurs orthographiques et grammaticales. Le logiciel dispose d'un modèle ELECTRA capable de détecter efficacement les erreurs en anglais. L'équipe de TAL (traitement automatique de la langue) de Druide désire mettre en place un système similaire pour la langue française. Dans le cadre de ce stage, le but est de créer un modèle ELECTRA capable de détecter les erreurs grammaticales en français. Pour ce faire, plusieurs approches ont été testées et les résultats des derniers modèles sont prometteurs. On remarque entre autre une hausse importante des performances en introduisant une tâche intermédiaire et en faisant une recherche d'hyperparamètres. Les méthodes n'ayant pas donné les résultats escomptés sont aussi discutées

Mots clés: Apprentissage automatique, Apprentissage profond, Apprentissage machine, Traitement de texte, Détection de mot manquants, Encodeur, Transformers, BERT, ELECTRA, Réseaux de neurones, Intelligence artificielle.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et des abréviations

MLM	Modélisation de langage avec masque, de l'anglais *Masked Language Modeling*
TAL	Traitement automatique du langage
NER	Reconnaissance d'entitées, de l'anglais *Named-Entity Recognition*
DDP	Parallélisme distribué des données, de l'anglais *Distributed Data Parallel*
TPE	Estimateur de Parzen à base d'arbres, de l'anglais *Tree-Structured Parzen Estimator*
LLM	Grand modèle de langue, de l'anglais *Large Language Model*

Remerciements

Je tiens remercier Joss Rakotobe, pour sa précieuse aide tout au long de mon stage. Je n'aurais pas pu demander un meilleur superviseur.
Je remercie aussi Momo pour son support moral constant.

Introduction

Le domaine du traitement automatique des langues connaît une explosion fulgurante de techniques, de jeux de données et de modèles permettant de résoudre de nouveaux problèmes. L'avènement des LLM à notamment mis à l'avant-plan les possibilités d'utilisation des modèles d'intelligence artificiels. Néanmoins, bon nombre de ces applications restent hors de portée des organisations désirant mettre en application des outils d'apprentissages automatique. En effet, la plupart des modèles de langues récents sont préentraînés sur des corpus majoritairement anglophones, avec des jetoniseurs spécialisés pour traiter le contenu anglophone. Ces deux facteurs limitent les modèles préentraînés disponibles ainsi que leur performance sur des tâches avec un corpus non anglophone.

Druide Inc. est une compagnie basée à Montréal dont le principal produit est Antidote, un logiciel de correction orthographique et grammaticale. Leur logiciel phare fait déjà usage de l'apprentissage profond pour leur moteur de correction en anglais, en plus d'utiliser un correcteur symbolique pour certains types d'erreurs. Le modèle utilisé en production pour la correction en anglais fait près de 2 corrections sur 3 et représente une part importante du moteur de correction. L'équipe de Druide désire mettre en place un modèle de correction similaire, mais adapté à la langue française. En particulier, ils désirent préentraîner un modèle ELECTRA avec un corpus et un jetoniseur francophones pour que le modèle puisse détecter les erreurs grammaticales présentes dans les textes francophones des utilisateurs d'Antidote.

Pour la réalisation du projet, nous disposons d'un jeu de données d'environ 40 GB de données non structurés, ainsi que d'un ensemble de phrases annotées. De plus, l'entraînement du modèle se fait localement sur une machine ayant accès à 3 NVIDIA RTX A4000, disposant chacune de 16 GB de mémoire VRAM.

Chapitre 1

Druide et ELECTRA

L'équipe de Druide dispose de deux modèles déjà en place pour la correction des erreurs. Néanmoins, leur modèle en anglais corrige une plus grande gamme d'erreurs. Druide désire améliorer leur moteur de correction en français à l'aide de l'apprentissage profond.

1.1. Contraintes

Le modèle doit être intégré dans le logiciel Antidote, principal produit de Druide. Or, le logiciel Antidote est déployé sur les ordinateurs personnels des usagers. Cela implique d'importantes contraintes quant aux ressources disponibles pour l'exécution du modèle, notamment en ce qui a trait à la consommation de mémoire. De plus, le logiciel Antidote se doit d'être rapide, puisqu'attendre plusieurs minutes pour la correction d'un texte volumineux dégrade la qualité de l'expérience pour les utilisateurs. En d'autres mots, le modèle doit être rapide durant l'inférence et ne peut pas consommer beaucoup de ressources sur l'appareil des usagers. Finalement, le logiciel Antidote cible deux systèmes d'exploitation: Windows et macOS. Le déploiement du modèle sur les machines des usagers se fait à l'aide des librairies ONNX[**4**] et CoreML. Il est donc nécessaire que le modèle soit supporté par les deux librairies. En résumé, nous avons des limites quant aux ressources disponibles durant l'inférence ainsi que des contraintes quant aux couches et modèles utilisables.

Ces contraintes ont poussé l'équipe du TAL de Druide à sélectionner des petits modèles Transformers[**11**] avec encodeur pour la tâche de détection des erreurs en anglais. Ces modèles contiennent environ 14 millions de paramètres et sont très rapides durant l'inférence.

1.2. Méthode de préentraînement ELECTRA

La méthode ELECTRA[**3**] est une méthode inspirée de la modélisation de langage avec masque (MLM), mais qui se veut plus efficace et rapide que le MLM. La méthode

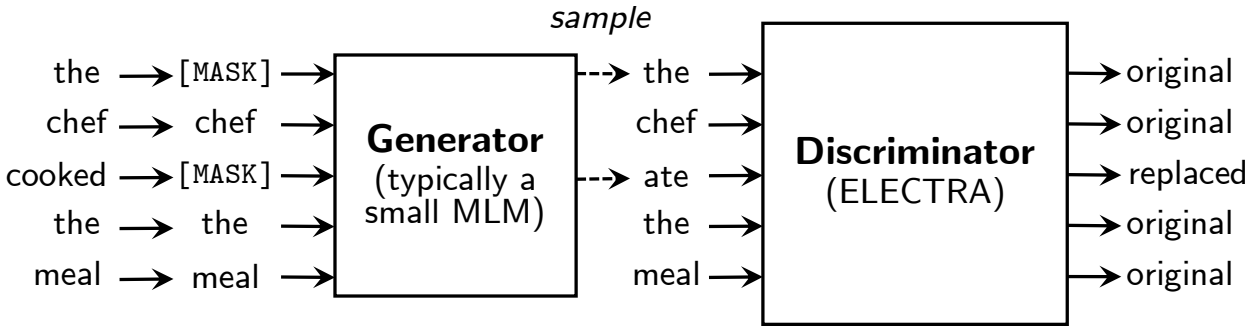

Fig. 1.1. Exemple de la méthode ELECTRA. Figure provenant de [**3**].

ELECTRA consiste à entraîner deux modèles: un petit modèle, appelé le générateur, et le modèle final, appelé le discriminant. Le générateur reçoit des jetons masqués aléatoirement et doit prédire quel était le jeton original situé à la position du masque. Les prédictions du modèle sont échantillonnées, de façon à obtenir une nouvelle séquence, potentiellement différente de la séquence originale. Le discriminant reçoit la nouvelle séquence et a pour tâche de prédire quels jetons sont corrompus et lesquels n'ont pas été modifiés par le générateur. Seul le discriminant est réutilisé pour l'affinage. La méthode est visualisée dans la figure 1.1.

La méthode ELECTRA a été choisie pour deux raisons: c'est une méthode de préentraînement similaire à la correction d'erreurs dans un texte, ce qui augmente la performance du modèle durant l'affinage. De plus, la méthode ELECTRA permet d'augmenter l'efficacité du préentraînement en atteignant des performances similaires aux performances d'un modèle entraîné avec du MLM, et ce, en moins d'itérations.

Trois éléments rendent l'entraînement du discriminant plus facile. Premièrement, le générateur a une capacité significativement plus faible que celle du discriminant. En effet, ce dernier contient en général 3 à 4 fois plus de paramètres (en excluant les couches de projections *embeddings*) que le générateur. De plus, les entrées du discriminant sont échantillonnées depuis la distribution engendrée par le générateur, au lieu de sélectionner les entrées les plus probables selon la distribution du générateur. Cela rend les prédictions du générateur moins précises. Finalement, les poids du générateur sont initialisés aléatoirement et ce dernier est entraîné en même temps que le discriminant, rendant la tâche de plus en plus difficile au fur et à mesure que le générateur s'entraîne. Ce facteur est l'une des raisons pour lesquelles le discriminant est capable d'apprendre pendant l'entraînement. Ces trois facteurs rendent la tâche du discriminant plus facile et permettent de générer des erreurs

similaires à ce que le modèle rencontrera en production.

1.3. Architecture ELECTRA

Le modèle ELECTRA utilise une architecture basée sur les modules d'encodeurs des Transformers [**11**]. L'usage d'une architecture basée sur les transformeur permet d'obtenir une représentation contextuelle pour tous les jetons d'une séquence. Cette avancée a été marquée par l'arrivée de BERT [**5**] (Bidirectional Encoder Representation from Transformers), un modèle Transformers. Ce modèle a été développé par Google en 2018 et comprend de nombreuses versions de différentes tailles entraînées sur différentes tâches. Les deux versions canoniques de BERT sont BERT-BASE et BERT-LARGE. Ces deux versions comprennent respectivement 12 couches et 24 couches d'encodeurs, chacune étant composée de 768 unités de large, divisées en 12 têtes d'attention multiples. Les auteurs d'ELECTRA ajoutent une nouvelle version plus petite basée sur BERT, dénommée ELECTRA-small. Celle-ci consiste en 12 couches de 256 unités de large et comporte environ 14 millions de paramètres. Ce modèle à base d'encodeur est composé de trois parties:

- Un **jetoniseur**, qui s'occupe de traiter le texte entrant et de le convertir un une séquence d'entiers.
- Une **couche de projection**, qui permet d'associer à chacun des jetons d'entrées une représentation vectorielle qui dépend de la position du jeton dans la séquence ainsi que du jeton lui-même.
- Un module d'**encodeurs**, qui permet d'obtenir une représentation contextualisée des entrées. Cette représentation est apprise et varie selon la tâche finale du modèle ainsi que le corpus utilisé pour entraîner le modèle.

Notre modèle ELECTRA utilise l'architecture ELECTRA-small.

1.4. Affinage pour la détection d'erreurs

Une fois le modèle ELECTRA préentraîné, il est nécessaire d'adapter le modèle pour que celui-ci soit en mesure de détecter efficacement les erreurs dans les textes des utilisateurs. Druide a développé une liste des différents types d'erreurs, permettant de classifier les différents types d'erreurs en de grandes catégories, telles que les erreurs de virgules, les erreurs de mots manquants, les erreurs d'accord du nom, etc. Cette liste contient plusieurs milliers de types d'erreurs. Néanmoins, dans notre cas, seuls 750 types d'erreurs doivent être détectés. Chaque erreur fait partie d'une des grandes catégories nommées ci-haut, et bon nombre de ces erreurs ont une sous-catégorie, précisant encore plus le contexte associé à l'erreur. La détection d'erreur est modélisée comme une tâche de détection d'entité nommée

(NER), dans laquelle chaque jeton dispose d'une classe. Les classes d'erreurs sont représentée avec un identifiant, tandis que la classe représentant l'absence d'erreurs est représentée par l'identifiant O. Le modèle a comme objectif de spécifier la classe de chaque jeton de la séquence. Le schéma $IOB2$ [**10**] est utilisé pour représenter sans ambiguïté les jetons contigus contenus dans la même erreur.

1.5. Infrastructures en place

Notre tâche principale consistait à préentraîner un modèle ELECTRA. Or, un modèle ELECTRA est déjà utilisé pour la tâche de correction en anglais. Ce dernier n'a pas été préentraîné par Druide. En effet, la librairie Transformers[**13**] permet un usage libre de différents modèles ELECTRA préentraînés. De plus, il existe quelques modèles ELECTRA préentraînés sur des corpus francophones. Cependant, aucun d'entre eux ne respecte nos contraintes de tailles et de vitesse. Il est donc nécessaire d'entraîner une modèle à partir d'un initialisation aléatoire.

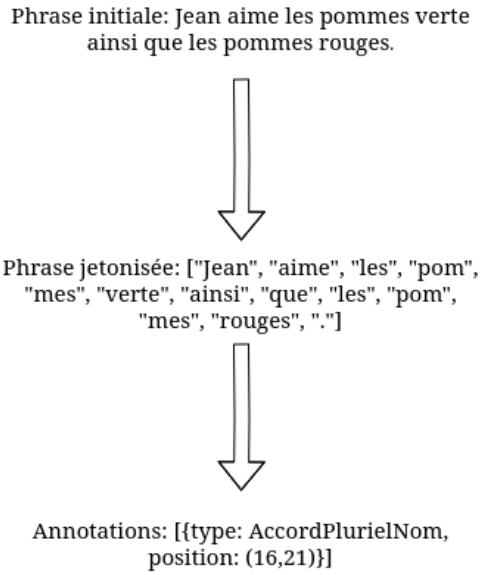

Fig. 1.2. Exemple de texte annoté. Les annotations se font au niveau des positions des res et ne sont donc pas spécifiques au jetoniseur

Nous disposons de deux corpus déjà préparés pour préentraîner et affiner un modèle Electra. Le corpus de préentraînement est une collection de textes non structuré provenant de nombreuses sources, notamment des manuels, des articles de blogues, des livres et des sites internet. Ce corpus de préentraînement est appelé corpus des Combis et représente 40

gigaoctets (Go) de données et 7 milliards de jetons. C'est un corpus deux fois plus grand que le corpus de préentraînement utilisé pour le préentraînement par Google du modèle ELECTRA de même taille. Pour l'affinage, Druide dispose d'un corpus contenant près de 100000 annotations sur des textes francophones. Ces annotations sont fournies par Druide et proviennent d'équipes de linguistes et d'annotateurs corrigeant des textes et classifiant les erreurs qu'ils y trouvent en fonction des types d'erreurs proposés par Druide. Un exemple d'annotation est donné dans la figure 1.2.

Chapitre 2

Entraînement de modèles initiaux

Le préentraînement d'un modèle de langue se fait en trois étapes. Il est nécessaire de prétraiter les données, de sélectionner un jetoniseur adapté à la tâche ainsi que d'entraîner le modèle sur la tâche de préentraînement.

2.1. Normalisation des données et entraînement d'un jetoniseur

La normalisation consiste à réduire le nombre de rs différents contenus dans le corpus. C'est une étape importante, puisqu'elle permet de réduire la taille du vocabulaire du jetoniseur sans pour autant perdre des éléments syntaxiques. Notre processus de normalisation consistait à transformer tous les caractères d'espacement (espaces insécables, tabulations, *U+2002*, etc.) en un même caractère d'espace *U+0020*. Notre processus de normalisation consiste aussi à transformer tous les guillemets (guillemets français, guillemets informatiques, etc.) en guillemets anglais, à retirer les espaces en trop et à modifier les types d'apostrophes pour que ceux-ci soient uniformes. La normalisation modifie aussi les espacements entre certains mots. Ainsi, le texte "11 ème étage" devient "11ème étage".

Une fois le texte normalisé, il est possible d'entraîner un jetoniseur adapté à la tâche. En l'occurrence, nous avons initialement choisi le jetoniseur WordPiece [**15**]. C'est le jetoniseur choisi par les auteurs de l'article ELECTRA et il est actuellement utilisé en production chez Druide. Il répond donc à nos contraintes. Pour l'entraînement du jetoniseur, nous utilisons le corpus des Combis après normalisation, comprenant environ 40 GO de données. Les hyperparamètres sélectionnés pour le jetoniseur WordPiece ainsi que pour la phase de préentraînement sont indiqués dans le tableau 2.1.

2.2. Préentraînement initial

Nous préentraînons le modèle à l'aide de l'implémentation originale d'ELECTRA. Celle-ci est disponible au dépôt git suivant: `https://github.com/google-research/electra`. Notre machine dédiée à l'entraînement dispose de 3 GPUs de 16 GB de VRAM chacun. Or, l'implémentation originale utilise moins de 16 GB de VRAM et ne permet pas l'usage de DDP, limitant la vitesse d'entraînement ainsi que la taille de lot (*batch size*). Pour tirer parti au maximum des 3 GPUs disponibles, nous avons initialisé aléatoirement trois modèles ELECTRA et avons changé l'ordre des données. En entraînant trois modèles sur le même jeu de données modifié, nous espérions être en mesure d'appliquer la méthode de la Soupe [**14**] une fois les modèles affinés et faire un meilleur usage des ressources computationnelles à notre disposition. Les hyperparamètres utilisés durant l'entraînement des trois modèles sont décrits dans le tableau 2.1. Nous avons divisé par 4 la largeur du générateur, tel que recommandé dans l'article original d'ELECTRA [**3**]. Nous avons suivi de près les hyperparamètres énumérés dans l'article ELECTRA.

hyperparamètres	Discriminant	Générateur
Nombre de couches	12	12
Taille des couches cachées	256	64
Taille des couches de projection	128	128
Nombre de têtes d'attention	4	1
Taille du vocabulaire	30522	30522
Ignore la casse	oui	oui
% des jetons masqués	-	15
Taux d'apprentissage	5e-4	5e-4
Poids de la perte du générateur	-	1
Poids de la perte du discriminant	50	-
Nombre d'itérations de *warmup*	10000	10000

Tableau 2.1. Hyperparamètres utilisés pour l'entraînement des modèles

Une fois le préentraînement complété, nous avons collecté les résultats d'évaluation de nos trois modèles sur la tâche d'ELECTRA. Les résultats sont disponibles dans le tableau 2.2.

Malgré le changement de l'ordre des données, aucun des trois modèles n'a performé significativement différemment des autres modèles. Il était donc possible que la méthode Soupe [**14**] permette d'augmenter la performance de nos modèles. La méthode Soupe est discutée plus en détail dans la section 4.

Métrique	Modèle 1	Modèle 2	Modèle 3
Exactitude du discriminant	0.952	0.949	0.950
AUC du discrimiantn	0.934	0.929	0.934
Perte du discriminant	0.135	0.142	0.138
Précision du discriminant	0.801	0.794	0.793
Rappel du discriminant	0.469	0.449	0.483
Perte totale	8.967	9.411	9.276
Exactitude du générateur	0.570	0.562	0.550
Perte du générateur	2.216	2.308	2.342
Exactitude du générateur	0.469	0.458	0.448

Tableau 2.2. Résultats d'évaluation des trois modèles sur la tâche ELECTRA

Modèle	Précision	Rappel	F0.5
Modèle initial 1	39.59	23.84	34.97
Modèle initial 2	40.22	24.36	35.63
Modèle initial 3	39.85	24.88	35.57

Tableau 2.3. Résultats des modèles préliminaires sur corpus d'évaluation de la tâche de correction

2.3. Premiers modèles affinés

Une fois les trois premiers modèles initiaux préentraînés, nous avons entraîné ces trois modèles sur la tâche de détection des erreurs en français. Les meilleures métriques enregistrées sont détaillées dans le tableau 2.3.

Les mesures F-0.5 obtenues sur l'ensemble de tests durant l'affinage sont contenues dans la figure 2.1. Nous utilisons la mesure F-0.5, dont la formule est donnée explicitement dans l'équation 2.3, comme principale métrique pour guider nos décisions. Un poids de 0.5 donne deux fois plus d'importance à la précision par rapport au rappel. Ce choix de métrique permet de prioriser les modèles qui minimisent les faux positifs et qui sont donc plus prudents dans leur détection des erreurs.

$$F_{0.5} = (1 + 0.5^2)\frac{\text{précision} \cdot \text{rappel}}{(0.5^2 \cdot \text{précision}) + \text{rappel}}$$

Tous les modèles ont été entraînés pour avec les hyperparamètres décrits dans le tableau 2.4.

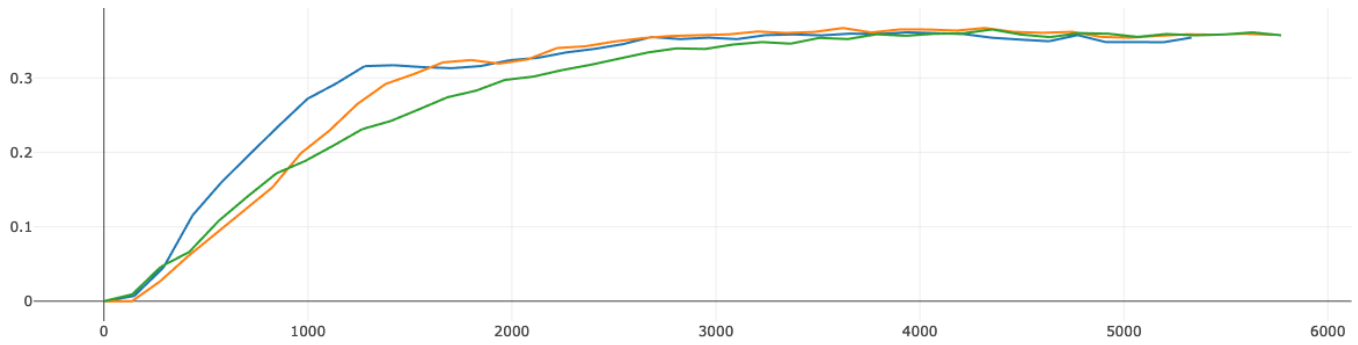

Fig. 2.1. Mesure F-0.5 des trois premiers modèles selon le nombre d'itérations sur données d'évaluation pour la détection d'erreurs en français.

Hyperparamètres	Discriminant
Taux d'apprentissage	1e-4
Optimisateur	AdamW
Betas de l'optimisateur	(0.9, 0.999)
Weight decay	0.01
Early stopping	oui
Nombre maximal d'époques	100
Taille de lots	128
Nombre d'itérations de *warmup*	10000
Scheduler	Linéaire

Tableau 2.4. Hyperparamètres utilisés pour l'affinage des modèles initiaux

2.4. Entraînement d'un modèle avec casse

Nos premiers modèles préentraînés sont limités en termes de performance par leur jetoniseur. En effet, le jetoniseur WordPiece ignore la casse, et certains types d'erreurs, tels que les majuscules pour les noms propres, nécessitent de connaître la casse des mots. Ce choix n'affecte que peu la performance du modèle durant le préentraînement, mais pose problème durant l'affinage sur la tâche de détection des erreurs. Nous avons donc entraîné un second jetoniseur WordPiece, cette fois-ci sensible à la casse, et avec maintenus tous les autres hyperparamètres identiques. Nous avons par la suite entraîné le modèle sur l'ensemble des

données, encore une fois avec l'ordre des données mélangé et avec les hyperparamètres du tableau 2.1. Les résultats du modèle avec casse ainsi que les résultats des modèles préliminaires sont contenus dans le tableau 2.5.

Métrique	Modèle 1	Modèle 2	Modèle 3	Modèle avec casse
Exactitude du discriminant	0.952	0.949	0.950	0.946
AUC du discrimiantn	0.934	0.929	0.934	0.931
Perte du discriminant	0.135	0.142	0.138	0.149
Précision du discriminant	0.801	0.794	0.793	0.795
Rappel du discriminant	0.469	0.449	0.483	0.471
Perte totale	8.967	9.411	9.276	10.31
Exactitude du générateur	0.570	0.562	0.550	0.499
Perte du générateur	2.216	2.308	2.342	2.848
Exactitude du générateur	0.469	0.458	0.448	0.403

Tableau 2.5. Résultats d'évaluation des trois modèles et du modèle avec casse sur la tâche ELECTRA

La performance du discriminant sensible à la casse est similaire à celles des autres discriminants. Cependant, le générateur sensible à la casse performe moins bien que les autres. Une explication possible est que la tâche du générateur, qui consiste à prédire le bon jeton à la position donnée, est plus difficile avec la casse, puisque les jetons contenant la casse sont souvent syntaxiquement identiques, mais sont traités comme étant différents durant l'évaluation.

Nous avons complété l'affinage du modèle en l'entraînant sur la tâche de détection des erreurs en français. Nous avons réutilisé les hyperparamètres des trois modèles précédents, contenus dans le tableau 2.1. Les résultats du modèle affiné et prenant en compte la casse sont contenus dans le tableau 2.6. Le modèle sensible à la casse performe mieux que les trois modèles initaux. Cette différence de performance s'explique par la présence d'erreurs basées sur la casse des mots dans le corpus d'affinage.

Les mesures F-0.5 obtenues sur l'ensemble de tests durant l'affinage pour le modèle avec casse sont contenues dans la figure 2.2.

2.5. Comparaison avec CAMEMBERT

Pour connaître le manque à gagner entre nos petits modèles ELECTRA et des modèles plus volumineux, nous avons affiné un plus gros modèle et avons comparé la performances

Modèle	Précision	Rappel	F0.5
Modèle initial 1	39.59	23.84	34.97
Modèle initial 2	40.22	24.36	35.63
Modèle initial 3	39.85	24.88	35.57
Modèle avec casse	42.59	25.81	37.69

Tableau 2.6. Résultats des modèles préliminaires sur corpus d'évaluation de la tâche de correction avec le modèle sensible à la casse

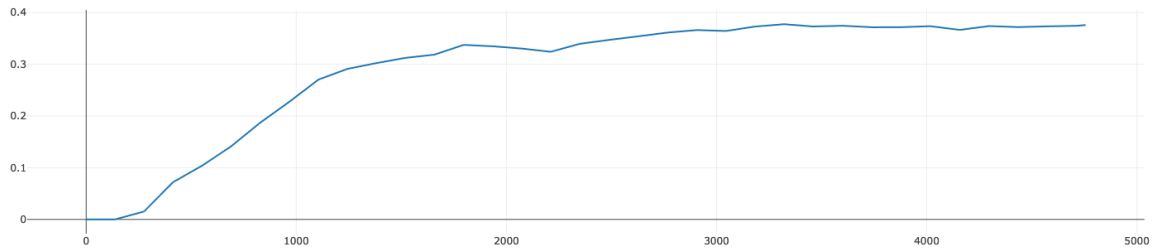

Fig. 2.2. Mesure F-0.5 du modèle avec casse selon le nombre d'itérations sur données d'évaluation pour la détection d'erreurs en français.

des modèles. Nous avons utilisé CAMEMBERT [**7**], un modèle basé sur BERT et préentraîné sur le corpus francophone OSCAR [**8**] à l'aide de la méthode MLM. CAMEMBERT-base possède plus de 100 millions de paramètres, le rendant près de 8 fois plus gros que nos modèles, qui, eux, sont basés sur l'architecture de ELECTRA-small. Ainsi, il est trop volumineux et lent pour être déployé dans Antidote. Néanmoins, il offre un possible plafond quant aux performances que nos modèles peuvent atteindre. De plus, le jetoniseur associé au modèle est sensible à la casse.

Le modèle CAMEMBERT a atteint une mesure F-0.5 maximale de 40.47, performant mieux que nos modèles initiaux et que le modèle sensible à la casse. La comparaison complète est disponible dans le tableau 2.7. L'évolution de la performance de CAMEMBERT au cours de son affinage est disponible dans la figure 2.3. Nous avons limité le nombre d'époques maximal à 100. Il aurait été intéressant de laisser le modèle s'entraîner pour plus d'époques et ainsi s'assurer que le modèle a atteint sa performance maximale.

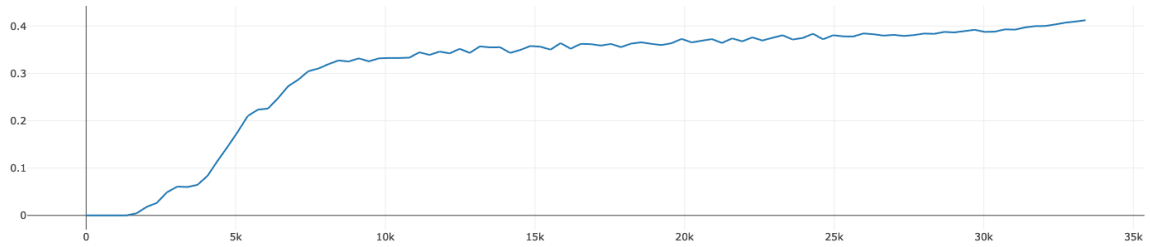

Fig. 2.3. Mesure F-0.5 de CAMEMBERT selon le nombre d'itérations sur données d'évaluation pour la détection d'erreurs en français.

Modèle	Précision	Rappel	F0.5
Modèle initial 1	39.59	23.84	34.97
Modèle initial 2	40.22	24.36	35.63
Modèle initial 3	39.85	24.88	35.57
Modèle avec casse	42.59	25.81	37.69
CAMEMBERT-base	44.74	29.30	40.47

Tableau 2.7. Résultats des modèles préliminaires sur corpus d'évaluation de la tâche de correction avec le modèle sensible à la casse et CAMEMBERT

Chapitre 3

Modèle final

Notre modèle final consiste en un modèle ELECTRA-small, faisant usage du jetoniseur de CAMEMBERT. Le jetoniseur de CAMEMBERT est un jetoniseur SentencePiece sensible à la casse et entraîné sur un corpus francophone. Les augmentations de performance les plus importantes proviennent des deux modifications suivantes: la recherche d'hyperparamètres, à l'aide de la librairie Optuna [**1**], et l'ajout d'une phase d'entraînement intermédiaire.

3.1. Recherche d'hyperparamètres

L'un des plus importants facteurs permettant d'augmenter la performance de notre modèle final est l'usage de méthode de recherche d'hyperparamètres. Nous avons utilisé la librairie Optuna, une librairie *Python* spécialisée pour la recherche d'hyperparamètres. En particulier, nous avons utilisé l'algorithme itératif *TPE* [**12**], [**2**]. Cet algorithme itératif part d'un ensemble d'hyperparamètres possibles et tente de trouver la combinaison d'hyperparamètres maximisant une métrique donnée. Dans notre cas, nous avons utilisé la mesure F-0.5 comme métrique à maximiser. Nous avons permis à l'algorithme TPE d'exécuter un maximum de 8 expériences, chacune faisant usage d'un différent ensemble d'hyperparamètres et entraînant nos modèles sur le corpus d'affinage. Pour toutes ces expériences, nous avons permis à l'algorithme *TPE* de sélectionner la valeur des hyperparamètres suivants:

- Taille des lots durant la phase d'entraînement
- *Weight decay* de l'optimisateur
- Taux d'apprentissage de l'optimisateur

Cette recherche d'hyperparamètres requiert un affinage complet du modèle pour chaque expérience et est donc trop dispendieuse pour être utilisée à grande échelle. Néanmoins, la recherche d'hyperparamètres a permis d'établir un nouveau record sur la tâche de correction en français, augmentant la F-mesure à 41,23. Cela représente un gain de 3.54 points de F-mesure.

3.1.1. Note sur la qualité des données d'affinage

La recherche d'hyperparamètres a révélé une tendance intéressante: les modèles gagnent à avoir une très petite taille de lot (*batch size*) lors de leur entraînement. En effet, les modèles performant le mieux ont obtenu leur meilleure performance en utilisant une taille de lot de 8. Cette observation offre un contraste drastique entre les méthodes d'entraînement contemporaines pour les modèles de langue, qui font souvent usage d'une très grande taille de lot. Une raison souvent donnée pour expliquer les meilleures performances des modèles entraînés avec une grande taille de lot est qu'une taille de lot plus grande offre une plus grande résistance quant aux exemples bruités, contenant des erreurs ou plus généralement de mauvaises qualités. Or, nous pensons que la meilleure performance des modèles utilisant une petite taille de lot s'explique par la qualité des données annotées. En effet, les exemples contenus dans le corpus de test sont annotés par des humains formés pour une telle tâche, et les textes sont basés sur des corpus filtrés des humains, bien souvent des linguistes. Il en résulte un corpus de très haute qualité contenant très peu d'exemples n'augmentant pas la performance finale du modèle.

3.2. Phase d'entraînement intermédiaire

Le plus gros gain de performance enregistré durant l'entraînement, en termes de mesure F-0.5 est l'ajout d'une phase d'entraînement intermédiaire. Cette phase additionnelle a rendu l'entraînement plus stable, plus rapide et a augmenté la performance du modèle final.

3.2.1. Collecte et extraction des données

Nous avons été en mesure d'augmenter les données annotés à l'aide des corrections des utilisateurs. En effet, Antidote dispose d'une version Web. Celle-ci est établie depuis de nombreuses années et collecte les textes anonymisés des utilisateurs. Cette base de données comporte notamment les phrases anonymisées entrées par les utilisateurs d'Antidote Web, ainsi que les corrections proposées par Antidote qui ont été acceptées par les utilisateurs. Ainsi, les erreurs détectées par Antidote qui ne sont pas réellement des erreurs (c'est-à-dire les faux positifs) sont potentiellement ignorées par les utilisateurs et ne font donc pas partie du nouveau corpus. Ce corpus ne contient pas toutes les erreurs contenues dans la liste de Druide, mais contient des exemples qui ont été indirectement vérifiés par des humains.

Après l'extraction des données, nous avons traité ces nouvelles données pour que ces derniers soient compatibles avec notre méthodologie d'entraînement. Nous avons notamment nettoyé le corpus en retirant les corrections ne faisant pas partie de la liste de corrections

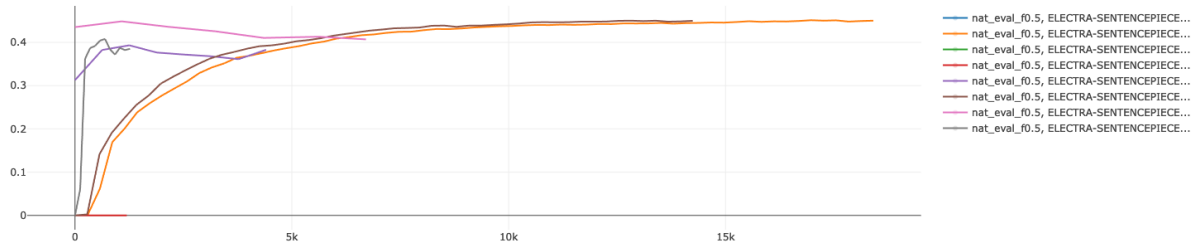

Fig. 3.1. Mesures F-0.5 des modèles ELECTRA avec SentencePiece sur la détections des erreurs en français, avec l'ajout de la tâche intermédiaire. Les différentes courbes correspondent à différentes expériences faites à l'aide d'Optuna. L'axe des 5 représente le nombre d'itérations et l'axe horizontal représente la mesure F-0.5.

utilisées dans l'affinage. Ce nouveau corpus contient un total de 3.5 millions de phrases annotées provenant des utilisateurs et corrigés par Antidote. Ainsi, ce dernier est 35 fois plus grand que le corpus d'affinage initial, qui contient 100000 corrections faites par des linguistes et des annotateurs.

3.2.2. Résultat de la phase intermédiaire

Nous avons utilisé ce nouveau corpus comme une tâche intermédiaire, exécutée après le préentraînement, mais avant l'affinage. Cette nouvelle phase intermédiaire s'est avérée être très importante pour la performance du modèle sur la correction des erreurs en français. En effet, nous avons observé une augmentation de près de 5 points de mesure F-0.5 ainsi qu'une hausse importante dans la vitesse à laquelle les modèles convergent, notamment dans les expériences de recherche d'hyperparamètres. La figure 3.1 illustre ce gain de performance.

Il est intéressant de noter la diminution du temps avant d'atteindre la convergence. En effet, les modèles initiaux (figure 2.1) débutent leur affinage avec une mesure F-0.5 de 0, tandis que le modèle final débute son affinage avec une mesure F-0.5 de près de 40. De plus, le nouveau modèle obtient une meilleure performance finale, comme indiqué dans le tableau des résultats 3.1.

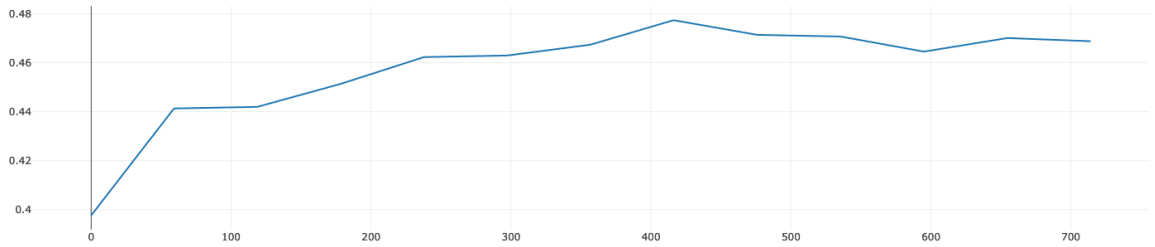

Fig. 3.2. Courbe de la mesure F-0.5 durant l'affinage du modèle final. Ce dernier correspond au meilleur modèle et utilise l'architecture ELECTRA. De plus, ce modèle a été entraîné avec la phase intermédiaire et utilise le jetoniseur SentencePiece.

Modèle	F0.5
Modèle initial 1	34.97
Modèle initial 2	35.63
Modèle initial 3	35.57
Modèle avec casse	37.69
CAMEMBERT-base	40.47
Modèle final (avec Optuna,sans phase intermédiaire)	41.23
Modèle final (avec Optuna et phase intermédiaire)	46.39

Tableau 3.1. Résultats finaux des modèles sur corpus d'évaluation de la tâche de correction

Chapitre 4

Autres techniques

La plupart des techniques mises en œuvre durant ma recherche n'ont pas donné les résultats escomptés. Cette section contient les résultats de ces différentes techniques qui n'ont pas amélioré la performance des modèles. Ces techniques y seront brièvement abordées.

4.1. Soupe

Nous avons appliqué la méthode Soupe (de l'anglais *Soup*) [**14**], qui consiste en une somme pondérée des poids des paramètres des différents modèles. Cette méthode simple permet d'augmenter la performance de modèles affinés sans pour autant faire usage de beaucoup de ressources et sans augmenter le temps d'inférence. L'intuition derrière cette technique est qu'un meilleur modèle se trouve souvent entre plusieurs modèles affinés. Donc, si nos modèles utilisés pour faire la Soupe sont à proximité d'un minimum local pour une fonction de perte donnée, alors la moyenne arithmétique des poids des modèles pourrait résulter en un modèle dont la perte est plus proche de l'optimum local.

Nous avons fait une Soupe à partir de nos 3 premiers modèles affinés, décrits dans la section 2.3. Nous avons par la suite affiné notre Soupe sur la tâche de détection des erreurs. Malheureusement, notre modèle n'a jamais été en mesure de converger durant l'affinage. Une explication possible est qu'au moins un de nos trois modèles affinés convergeait vers un optimum local différent des optimums des autres modèles. Ainsi, l'initialisation du modèle Soupe était éloignée des optimums locaux des modèles et résulte en un modèle incapable d'apprendre efficacement.

4.2. Nettoyage des données

Une part essentielle de l'entraînement est de fournir des données de qualité à notre modèle. Or, une partie importante du corpus de préentraînement est de piètre qualité. En

effet, les données proviennent de plusieurs types de sources différentes: livres, revues, article de blogue et site internet. Toutes ces sources ne sont pas sans erreurs ou ne contiennent pas nécessairement des textes avec un niveau de langue suffisant. C'est pourquoi retirer les exemples de piètre qualité est importants. Or, le nettoyage d'un corpus de grande taille nécessite l'usage d'heuristiques de sélection. Pour la création du pipeline de nettoyage, nous avons utilisé la librairie *Datatrove* de *HuggingFace* [**9**] et l'avons modifié pour qu'elle convienne à notre usage.

Nous avons appliqué principalement 4 filtres, dont l'ordre d'apparition est le suivant:

- Filtre de longueur
- Filtre de langue
- Filtre de Gopher
- Filtre de C4

Le filtre de longueur retire les exemples qui contiennent moins de 6 mots et le filtre de langue retire les exemples qui ne sont pas en français. Ce dernier fait usage d'un modèle FastText [**6**] pour détecter la langue du texte. Le filtre Gopher est plus complexe et utilise plusieurs heuristiques pour filtrer le contenu du corpus. Si l'une des conditions suivantes est remplie, alors l'exemple sera retiré du corpus propre.

- La longueur de l'exemple est supérieure à 100000 mots et inférieure à 5 mots
- La proportion de lettres parmi l'exemple est inférieure à 70%
- La proportion de symboles parmi les mots est supérieure à 10%
- Le nombre de *stop words* est inférieur à 2

Finalement, le filtre C4 retire les exemples contenant du JavaScript, le terme *Lorem Ipsum*, le symbole '{' ou encore les exemples contenant des conditions d'utilisations.
Les filtres ont retiré près de 23% du contenu total initial du corpus: parmi les 314 907 209 exemples initiaux, il en reste 242 710 245 dans le corpus nettoyé. Les différents filtres ont eu un impact variable sur le résultat final:

- filtre de longueur: retiré 38 585 180 exemples
- filtre de langue: retiré 8 283 109 exemples
- Gopher: retiré 25 228 530 exemples
- C4: retiré 14 741 exemples

Après le nettoyage du corpus, nous avons entraîné une nouvelle fois un modèle, mais cette fois-ci sur le corpus restreint. Nous avons utilisé les mêmes hyperparamètres que le meilleur modèle décrit au chapitre 3 durant le préentraînement et la phase intermédiaire, puis avons effectué une recherche d'hyperparamètres à l'aide d'Optuna. La figure 4.1 montre les courbes de mesure F-0.5 durant la recherche d'hyperparamètres sur la tâche de détection

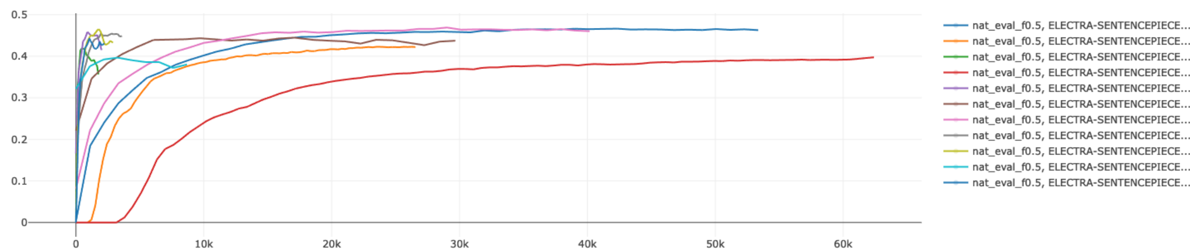

Fig. 4.1. Mesure F-0.5 durant la recherche d'hyperparamètres après avoir préentraîné sur le corpus nettoyé. Aucun des modèle n'atteint la performance du modèle final (3.2)

des erreurs en français.

Nous sommes incertains de la cause de la baisse de performance après le nettoyage des données. Une hypothèse est que nos heuristiques sont trop sévères. Par exemple, le filtre de longueur retire les exemples de moins de 6 mots, ce qui correspond à retirer plus de 38 millions d'exemples. Il en résulte que notre modèle n'aura jamais été entraîné sur des exemples contenant moins de 6 mots durant l'affinage. Plus de tests sont nécessaires pour vérifier la pertinence de nos heuristiques. En particulier, il serait pertinent de s'assurer que le filtre de longueur n'est pas trop sévère.

Chapitre 5

Conclusions

Durant ce stage, je me suis familiarisé avec les principales étapes du développement d'un système en traitement automatique de la langue. J'ai travaillé sur le nettoyage et traitement des données, le prototypage de différents modèles ainsi que l'évaluation des modèles d'apprentissage automatique. De plus, j'ai appris à connaître l'écosystème d'apprentissage automatique en *Python* ainsi que les modèles considérés comme l'état de l'art en traitement automatique des langues.

J'ai obtenu de bons résultats, notamment en ajoutant une phase intermédiaire durant l'entraînement et en effectuant une recherche d'hyperparamètres. Notre meilleur modèle est même en mesure de dépasser la performance de CAMEMBERT, un modèle disposant de presque 8 fois plus de paramètres. Nous pensons qu'une avenue d'amélioration du modèle consisterait à faire usage de grands modèles de langues dans le but de générer des données artificielles de bonne qualité. En particulier, une stratégie qui pourrait être intéressante pour augmenter la performance du modèle serait d'utiliser un grand modèle de langue pour générer des phrases comportant des erreurs dans le but d'augmenter la taille du corpus d'entraînement. Il serait particulièrement intéressant de générer des données artificielles pour les types d'erreurs avec lesquels notre modèle a encore de la difficulté. Cela permettrait de réduire le coût et le temps de génération des phrases artificielles et permettrait d'avoir un modèle plus performant sur tous les types d'erreurs présent dans la liste des erreurs corrigées par Antidote.

Références

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, et Masanori Koyama, *Optuna: A next-generation hyperparameter optimization framework*, The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 2623–2631.

[2] James Bergstra, Rémi Bardenet, Yoshua Bengio, et Balázs Kégl, *Algorithms for hyper-parameter optimization*, Advances in Neural Information Processing Systems (J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, et K.Q. Weinberger, eds.), vol. 24, Curran Associates, Inc., 2011.

[3] Kevin Clark, Minh-Thang Luong, Quoc V. Le, et Christopher D. Manning, *Electra: Pre-training text encoders as discriminators rather than generators*, 2020.

[4] ONNX Runtime developers, *Onnx runtime*, `https://onnxruntime.ai/`, 2021, Version: x.y.z.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, et Kristina Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019.

[6] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, et Tomas Mikolov, *Fasttext.zip: Compressing text classification models*, arXiv preprint arXiv:1612.03651 (2016).

[7] Louis Martin, Benjamin Muller, Pedro Javier Ortiz Suárez, Yoann Dupont, Laurent Romary, Éric de la Clergerie, Djamé Seddah, et Benoît Sagot, *Camembert: a tasty french language model*, Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 2020.

[8] Pedro Javier Ortiz Suárez, Benoît Sagot, et Laurent Romary, *Asynchronous pipelines for processing huge corpora on medium to low resource infrastructures*, , Proceedings of the Workshop on Challenges in the Management of Large Corpora (CMLC-7) 2019. Cardiff, 22nd July 2019, Leibniz-Institut für Deutsche Sprache, 2019, pp. 9 – 16 (en).

[9] Guilherme Penedo, Hynek Kydlíček, Alessandro Cappelli, Mario Sasko, et Thomas Wolf, *Datatrove: large scale data processing*, 2024.

[10] Lance A. Ramshaw et Mitchell P. Marcus, *Text chunking using transformation-based learning*, 1995.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, et Illia Polosukhin, *Attention is all you need*, 2023.

[12] Shuhei Watanabe, *Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance*, 2023.

[13] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, et Alexander M. Rush, *Transformers: State-of-the-art natural language processing*,

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (Online), Association for Computational Linguistics, October 2020, pp. 38–45.

[14] Mitchell Wortsman, Gabriel Ilharco, Samir Yitzhak Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, et Ludwig Schmidt, *Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time*, 2022.

[15] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, et Jeffrey Dean, *Google's neural machine translation system: Bridging the gap between human and machine translation*, 2016.