

Génération de code HTML à partir de screenshots de design

Yoann Gauthier
CentraleSupelec

yoann.gauthier@student.ecp.fr

Abstract

Passer d'un design à du code de site web est toujours une étape longue est peu intéressante pour les développeurs web. A l'aide d'une approche basée sur des réseaux convolutifs et récurrents, nous générons dans ce projet le code HTML correspondant à un screenshot de design donné. Pour des raisons de complexité, notre approche se restreint aux cas de sites internet ayant des designs proches (ie utilisant une feuille de style CSS commune). Sur ces cas restreints, nous obtenons une précision entre 50% et 60% sur un score BLEU.

Le code associé à ce projet est disponible sur un repo Github public ¹ L'entraînement du réseau prend environ 1h sur Google Colab (avec une carte GPU Tesla K80).

1. Introduction

1.1. Contexte

De nos jours, l'implémentation de site web nécessite le travail coordonné de plusieurs équipes qui ont des domaines de compétences différentes (design, front, back, base de données, admin système ...), créant de fait des délimitations sur le travail attribué à chacun. Une de ces frontières est notamment la mise en place du design d'un site par les développeurs frontend. Les équipes de design se chargent généralement de réaliser des maquettes, de concevoir l'expérience utilisateur et de faire les choix de design, que les développeurs doivent ensuite implémenter eux-même. Cette étape est souvent fastidieuse (il faut faire attention à respecter chaque marge, couleur, typographie, ...) et peu intéressante car ne correspondant pas à la logique de l'application web. La génération automatique de ce code de structuration et d'affichage de la page prends alors tout son sens.

Ici l'exemple est pris pour du développement web (prépondérant de nos jours), mais des problèmes similaires

se posent pour le développement mobile ou même de logiciels.

La génération de code HTML ou CSS n'est pas une idée nouvelle (des logiciels comme Dreamweaver [6] le faisaient déjà au début des années 2000), mais requérait une action humaine et limitait drastiquement les possibilités de design. Les réseaux de neurones permettent aujourd'hui de remédier à ces problèmes en générant directement le code HTML et CSS correspondant à des designs donnés (sous forme de screenshot), qui peuvent être réalisés par des équipes de designers sous des logiciels spécifiques (Adobe InDesign par exemple).

Pour des problèmes de longueur d'entraînement (5h sur une carte Tesla K80 disponible avec des offres gratuites ou étudiantes), nous nous restreignons dans ce sujet à la génération de code HTML (la structure de la page donc), qui nécessite sensiblement moins de temps (1h sur une Tesla K80).

1.2. Idée générale de l'approche

Notre réseau de neurones va interagir avec 2 types d'éléments, et nécessite pour cela 2 types de structures différentes (voir figures 1 et 2):

- Les screenshots de la page designée : Le premier problème est donc de la détection de forme, celle des éléments qui composent la page. Pour cela on utilise un réseau de type CNN.
- Le code de la page HTML (pour l'entraînement) ou le code déjà généré avant l'étape courante (pour le test). L'analyse de la structure de la page se rapproche de l'analyse textuelle : on définit un vocabulaire (celui de l'ensemble des balises), puis on analyse séquentiellement le "contexte" (la séquence de balises HTML précédant la balise que l'on essaye de prédire) afin de générer une balise.

2. Etat de l'art

Une des meilleures popularisation du code proposé par Beltramelli, utile pour bien comprendre la structure du

¹Fork du projet Screenshot-To-Code : <https://github.com/yoagauthier/Screenshot-to-code-in-Keras>

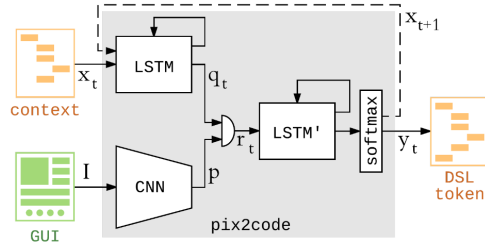


Figure 1. Modèle du réseau lors de la phase d'entraînement (issu de l'article [1])

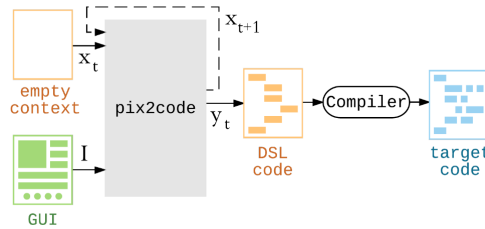


Figure 2. Modèle du réseau lors de la phase de test (issu de l'article [1])

réseau et les interactions entre les différents éléments est un article de blog de Emil Wallner [5].

L'évolution logique de la génération de code (web ou mobile) à partir de maquettes de design sous format image (en png notamment) est la génération de code directement à partir de dessins sur papier ou tableau. On mentionne à ce titre le projet SketchToProduct d'Airbnb [8] ou les services proposés par la startup Uizard [4].

Avec ce genre de systèmes, les possibilités de test et d'itération pour les designers et développeurs sont grandement simplifiées et les cycles de conception de nouveaux produits considérablement accélérés.

3. Approche

Le problème est ici approché comme un problème de génération avec deux encodeurs, qui vont traduire l'image et le contexte (le code déjà généré), et un décodeur qui classe la balise à générer en sortie en fonction des features générées par les deux encodeurs.

3.1. Jeux d'entraînement et de test

Pour avoir des situations les plus proches possibles des cas réels, nous avons récupéré des templates HTML désignés à l'aide du framework CSS bootstrap [3] et préparé les sets associés. 5 templates ont constitué le set d'entraînement, et un le set de test. Chaque template est constitué d'environ 200 balises HTML, mais le Tokenizer détecte entre 300 et 500 éléments par fichier, car il tokenize

parfois des paramètres des balises.

Ce set est différent du set d'origine proposé par Wallner [5], ce qui permet de comparer les résultats.

3.2. Construction du vocabulaire

Avant toute chose, il nous faut construire un vocabulaire de l'ensemble des balises HTML pouvant être générées. 3 possibilités se présentent :

- Construire le vocabulaire de l'ensemble des balise HTML existantes à partir de documentations officielles. Non seulement ce dictionnaire pourrait être long et difficile à construire, mais il contiendrait surtout un très grand nombre de balises possibles (dont certaines peuvent être exotiques ou rarement utilisée), d'autant plus si l'on souhaite générer un ensemble HTML / CSS.
- Réduire le vocabulaire à un ensemble restreint de balises prédéfinies. C'est l'approche du papier original [1] qui définit un DSL (Domain Specific Language) qui est convertible en code HTML / CSS. On a donc un nombre de balises plus restreint, mais les possibilités de générations de designs sont donc aussi plus restreint (à un style spécifique par exemple). Dans une approche où l'utilisateur final souhaiterait prototyper rapidement son interface web, un DSL restreint n'est pas trop gênant
- Construire directement le vocabulaire à partir des sets d'entraînement : à partir d'un Tokenizer, on récupère l'ensemble des balises existantes dans les fichiers HTML de train. Cette approche est la plus flexible et rapide, permet de ne pas générer de vocabulaire superflu (donc pas de paramètres superflus dans le modèle), mais possède l'inconvénient que le réseau ne pourra s'adapter à des balises jamais non présentes dans les sets de train. C'est l'approche que nous suivons par la suite.

3.3. Entrées et sorties de chaque élément

En entrée de l'encodeur de screenshot, nous avons donc des images sous format PNG, et à la sortie de cet encodeur, nous récupérons des features map générées par le réseau convolutif (sans couche dense de sortie), appelée screenshot features, qu'on peut voir par exemple sur la figure 3.

En entrée de l'encodeur de contexte, nous avons du code HTML, qu'on convertit en une séquence de balises successives. Cette séquence de balise est convertie en "word-embedding", une représentation qui attribue à chaque token de notre vocabulaire un nombre. Le word-embedding est appris par le réseau. A la sortie de cet encodeur (appelé "context encoder") des "markup features" qui représentent les balises précédemment générées.



Figure 3. Exemple de feature map générée par le InceptionResNet

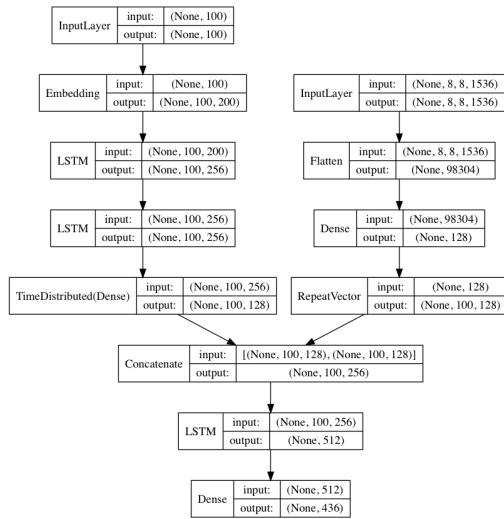


Figure 4. Architecture proposée par E. Wallner et prise comme base pour ce projet

On concatène ensuite ces features et on les envoie dans le décodeur. La sortie du décodeur est de manière assez classique un vecteur représentant la probabilité de la classe de la balise à générer dans chacun des éléments du vocabulaire.

Dans le cas de la génération (par opposition à l'entraînement), on ajoute la balise de sortie à la séquence de balises d'entrée du code HTML, ce qui permettra de générer la balise suivante correspondante (sortie x_{t+1} sur la figure 1).

3.4. Architecture du réseau

Pour un schéma global de l'architecture et des dimensions des paramètres, voir la figure 4

3.4.1 Encodeur d'images

Dans notre cas, il est possible d'utiliser un réseau pré-entraîné sur ImageNet pour générer les features map correspondant au screenshot d'entrée. Ici nous utilisons InceptionResNet qui permet de générer environ 1500 features map de taille 8x8 pour chaque screenshot.

Plus loin dans cet article nous détaillons des méthodes de vision par ordinateur n'utilisant pas de réseaux de neurones pour remplacer de réseau convolutif lourd.

3.4.2 Encodeur de contexte

L'approche ici est similaires à celle du papier context2vec [2], à la différence que nous utilisons simplement 2 couches LSTM monodirectionnels. L'idée est que ces réseaux récurrents "apprennent" l'historique des balises générées les unes après les autres pour savoir quelles sont les balises ouvertes. Une contrainte pour l'apprentissage de ce contexte est de savoir combien d'éléments on enregistre, ie combien d'éléments on prend en compte dans la séquence de balise qu'on met en entrée de cet encodeur de contexte. Plus la page HTML sera longue (ie complexe, avec de nombreux éléments), plus on devrait avoir un historique important.

3.5. Décodeur de features et génération de token

Le décodeur est composé d'une couche LSTM pour générer la feature correspondante à la prochaine balise à générer et d'une couche MLP pour classifier la prédiction finale. On génère donc dans cette couche les tokens 1 par 1.

4. Expériences

4.1. Approche classique de vision par ordinateur

Afin de diminuer la complexité du réseau (et donc diminuer les temps d'entraînement et de test, principaux facteurs limitants à l'utilisation pratique de ce système), nous avons essayé de remplacer l'encodeur de screenshots (le réseau convolutif) par des approches classiques de vision par ordinateur à l'aide d'OpenCV.

4.1.1 Détection de points d'intérêts + échelle de tons

Pour commencer, nous avons essayer de détecter les points d'intérêts sur un screenshot à l'aide de la méthode SIFT. Ces points d'intérêts devraient délimiter les blocs de design ou les différents éléments de la page. La méthode SIFT nous fournit les localisations des points d'intérêts, le rayon des points, et 128 descripteurs pour chacun des 5. Pour compléter cette approche, on peut essayer d'appliquer des filtres pour détecter les éléments de chacune des couleurs.

Ceci implique de modifier légèrement la structure du réseau pour prendre en compte un vecteur de descripteurs

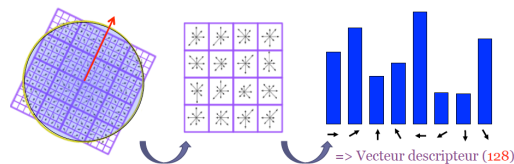


Figure 5. Structure d'un descripteur SIFT

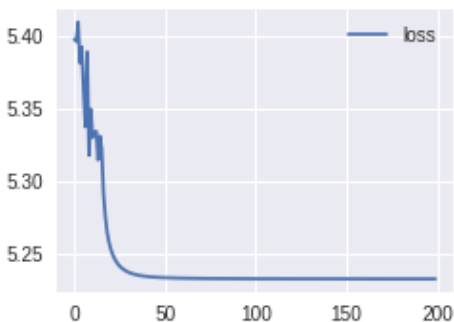


Figure 6. Fonction de coût lors de l'entraînement. Le réseau ne semble pas assez complexe et la fonction ne décroît pas suffisamment pour obtenir des résultats

à la place d'une feature map pour la sortie de l'encodeur de screenshots.

Cette approche permet de détecter environ 90 descripteurs de taille 128 et 25 tranches de couleurs, ce qui est un nombre de paramètres très faible comparé aux 1536 features map de taille 8x8 générées par le réseau InceptionResNet. Il n'est donc pas étonnant de s'apercevoir par la suite que le réseau n'arrive pas à apprendre suffisamment (sur le set de train, la loss function ne descend pas sous des valeurs $j=4$, voir 6), on manque probablement de paramètres et d'informations pertinentes (les descripteurs ne sont pas toujours correctement localisés).

Nous avons donc ensuite envisagé une approche moins naïve.

4.2. Détection des régions & downsampling

Plutôt que détecter des points sur une image caractérisée par des polices, des zones de couleur légèrement différentes de l'arrière plan ou des encadrés, nous avons donc essayé de détecter des régions.

On commence pour cela par thresholder les images pour récupérer les éléments indépendamment du background (le texte et les encadrés sont généralement de couleur différente sur un design de site web). On applique ensuite une dilata-tion pour "fusionner" les éléments textuels (on ne cherche pas à récupérer les lettres mais le rendu global). Enfin on récupère les contours de chaque élément de la page (qui correspondra donc à une balise HTML), ce qui nous permet

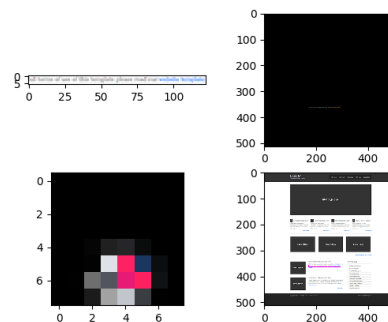


Figure 7. En haut à gauche : élément du screenshot découpé après détection des contours. En haut à droite : construction de l'image noire avec l'élément découpé. En bas à gauche : feature map générée par le rescaling avec convolution gaussienne (sur les 3 channels RGB). En bas à droite : élément du screenshot détecté et encadré

d'avoir la localisation de chaque élément.

Pour générer les features map correspondantes à chaque élément que nous avons détecté, on va générer une image noire avec une fenêtre laissant paraître l'élément détecté. On downsize ensuite chaque feature map (de 2000x2000 pixels à 8x8 pixel), en divisant par deux à chaque étape la taille de l'image avec une convolution. Le kernel de la convolution est une matrice Gaussienne :

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \quad (1)$$

Les résultats intermédiaires de chaque étape sont présentés sur la figure. 7.

La principal frein à cette approche est que le nombre d'éléments à détecter varie en fonction du screenshot, donc le nombre de features map également. Pour satisfaire les dimensions du réseau, on est alors obligé de prendre soit un nombre restreint de features map, soit de combler les vides avec des données random.

Cette approche donne cependant des résultats trop aléatoires : on a rapidement tendance à overfitter et le score BLEU sur les sets de test (détaillé plus bas) est très faible (de l'ordre de 2% - 4%). Bien qu'ayant plus de paramètres que la première approche, le fait que les features map ne soient pas apprises avec le réseau rend forcément cette approche moins précise que celle du réseau convolutif. Elle n'est donc pas concluante mais reste intéressante si on arrive à la complexifier.

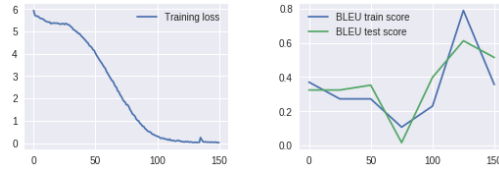


Figure 8. Evolution de la fonction de coût lors d'un entraînement, et calcul des scores BLEU associés sur un élément de l'ensemble de train et l'ensemble de test. On s'aperçoit de l'overfitting au delà de 125 epochs

4.3. Mesure des performances

Pour comparer le code généré au code HTML de la page d'origine, il est absurde de comparer directement la séquences de balises, car une seule inversion ou un décalage rendrait la page fautive. Comme proposé dans le travail de Wallner [5], nous utilisons la mesure BLEU [7] qui permet de mesurer une précision : le nombre de mots valides des sous-séquences de 4 mots. On utilise cette mesure au cours de l'entraînement (sur le set de test) pour vérifier que l'on sur-entraîne pas le modèle.

Nous avons mis du temps à comprendre pourquoi la mesure BLEU ne devenait pas aussi proche de 1 (ie que la précision restait faible) par rapport aux articles de référence. Wallner ne sépare pas les sets de train et de test, donc son taux élevé est cohérent, et lorsque nous avons pris soin de séparer les deux, la fonction de coût descendait bien lors de l'entraînement mais la généralisation était mauvaise. Ceci est dû au fait que les templates HTML utilisés ne se recoupent pas entièrement, alors que l'on construit notre vocabulaire de balises à partir de ces templates : si la balise n'a jamais été rencontrée dans les sets de train, elle ne peut pas être générée lors du test et la précision du test est mauvaise. Une fois ceci compris, nous avons créé un nouveau template de test à partir de parties de code des autres templates, ce qui a permis d'augmenter le score BLEU de 30% à 60%.

Sur la figure 9, on peut observer un résultat généré sur un des sets et le screenshot de départ. Les "gros" éléments ont été correctement générés, mais le résultat est beaucoup plus mitigé pour les simples listes de texte par exemple. On peut donc raisonnablement supposer que c'est plus l'encodeur de screenshots (le réseau CNN) qui est limitant ici, car il ne permet pas de détecter suffisamment les éléments "fins".

On constate également une différence de précision entre les deux dataset de templates : celui de Wallner permet un score BLEU plus élevé. Ceci est probablement dû au fait que les pages sont plus longues sur notre set (avec beaucoup de code concernant des animations sur des boutons, donc visuellement inutile), donc il est plus facile pour le réseau lors de l'apprentissage, de se "souvenir" des relations entre balises.



Figure 9. En haut : le screenshot de départ. En bas : le résultat du code HTML généré (score BLEU de 49% pour 150 epochs)

5. Extension de l'approche pour des cas réels

Pour réduire la complexité et le temps de calcul de ce problème, nous nous sommes restreints à la génération de balise HTML, sans le CSS associé, ce qui peut paraître assez absurde au premier abord, vu notre screenshot désigné ne sera pas retranscrit entièrement en code. Ici nous avons supposé une génération de code HTML "à style constant", c'est-à-dire que les sets de train et de test sont tous associés à un même fichier de style (fichier styles/layout.css). Pour éviter l'explosion du nombre de balise possibles ($nb_{balisehtml} * nb_{baliseccs}$), le mieux reste de passer par un langage de description intermédiaire (DSL) comme détaillé plus haut.

Il faudrait également pouvoir entraîner le modèle sur plus de codes / screenshots pour envisager plus de cas différents et complexifier le modèle pour augmenter la

précision. Dans son papier, Beltramelli [1] entraîne son propre réseau CNN au lieu d'utiliser un réseau pré-entraîné, ce qui est plus long (5h, en augmentant aussi le nombre de paramètres des LSTM) mais augmente la précision.

6. Conclusion

A l'aide d'une approche assez similaire au NLP (en ce qui concerne la représentation des entrées et la prise en compte du contexte), nous avons réussi à générer du code HTML comme nous pourrions générer des phrases. Après avoir mis en oeuvre le un modèle End to end (permettant un entraînement en une seule fois), nous avons essayé de réduire la complexité du modèle originellement proposé à l'aide d'approches de vision par ordinateur, malheureusement sans succès concluant : le temps de calcul gagné est trop faible au vu de la perte de précision sur la sortie générée.

Ce projet reste néanmoins pour nous de bon intérêt pédagogique car il aborde nombre de concepts transverses (word embedding, réseaux pré-entraînés) et a permis de mettre en valeur quelques problèmes classiques du deep learning (séparation train/test, overfitting, ...)

References

- [1] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962*, 2017.
- [2] O. Melamud, J. Goldberger, and I. Dagan. context2vec: Learning generic context embedding with bidirectional lstm. 2016.
- [3] D. Miller. Start bootstrap simple templates. <https://startbootstrap.com/template-categories/all/>, 2018.
- [4] Uizard. <https://uizard.io/>, 2018.
- [5] E. Wallner. Turning design mockups into code with deep learning. <https://blog.floydhub.com/turning-design-mockups-into-code-with-deep-learning/>, 2018.
- [6] Wikipedia. Adobe dreamwaever — Wikipedia, the free encyclopedia, 2018.
- [7] Wikipedia. Bleu — Wikipedia, the free encyclopedia, 2018.
- [8] B. Wilkins. Sketching interfaces — airbnb. <https://airbnb.design/sketching-interfaces/>, 2018.